

- 容器技术是继大数据和云计算之后又一热门技术，而且未来相当一段时间内都会非常流行
- 对 IT 从业者来说，掌握容器技术是市场的需要，也是提升自我价值的重要途径
- 每一轮新技术的兴起，无论对公司还是个人既是机遇也是挑战



Production-Grade Container Orchestration

每天5分钟 玩转Kubernetes

CloudMan 著

清华大学出版社

每天5分钟玩转Kubernetes

CloudMan 著

清华大学出版社





每天5分钟 玩转Kubernetes

CloudMan 著

清华大学出版社
北京

目录

[内容简介](#)

[前言](#)

[第1章 先把Kubernetes跑起来](#)

[1.1 先跑起来](#)

[1.2 创建Kubernetes集群](#)

[1.3 部署应用](#)

[1.4 访问应用](#)

[1.5 Scale应用](#)

[1.6 滚动更新](#)

[1.7 小结](#)

[第2章 重要概念](#)

[第3章 部署Kubernetes Cluster](#)

[3.1 安装Docker](#)

[3.2 安装kubelet、kubeadm和kubectl](#)

[3.3 用kubeadm创建Cluster](#)

[3.3.1 初始化Master](#)

[3.3.2 配置kubectl](#)

[3.3.3 安装Pod网络](#)

[3.3.4 添加k8s-node1和k8s-node2](#)

[3.4 小结](#)

[第4章 Kubernetes架构](#)

[4.1 Master节点](#)

[4.2 Node节点](#)

[4.3 完整的架构图](#)

[4.4 用例子把它们串起来](#)

[4.5 小结](#)

[第5章 运行应用](#)

[5.1 Deployment](#)

[5.1.1 运行Deployment](#)

[5.1.2 命令vs配置文件](#)

[5.1.3 Deployment配置文件简介](#)

[5.1.4 伸缩](#)

[5.1.5 Failover](#)

[5.1.6 用label控制Pod的位置](#)

[5.2 DaemonSet](#)

[5.2.1 kube-flannel-ds](#)

[5.2.2 kube-proxy](#)

[5.2.3 运行自己的DaemonSet](#)

[5.3 Job](#)

[5.3.1 Pod失败的情况](#)

[5.3.2 Job的并行性](#)

[5.3.3 定时Job](#)

[5.4 小结](#)

[第6章 通过Service访问Pod](#)

[6.1 创建Service](#)

[6.2 Cluster IP底层实现](#)

[6.3 DNS访问Service](#)

[6.4 外网如何访问Service](#)

[6.5 小结](#)

[第7章 Rolling Update](#)

[7.1 实践](#)

[7.2 回滚](#)

[7.3 小结](#)

[第8章 Health Check](#)

[8.1 默认的健康检查](#)

[8.2 Liveness探测](#)

[8.3 Readiness探测](#)

[8.4 Health Check在Scale Up中的应用](#)

[8.5 Health Check在滚动更新中的应用](#)

[8.6 小结](#)

[第9章 数据管理](#)

[9.1 Volume](#)

[9.1.1 emptyDir](#)

[9.1.2 hostPath](#)

[9.1.3 外部Storage Provider](#)

[9.2 PersistentVolume & PersistentVolumeClaim](#)

[9.2.1 NFS PersistentVolume](#)

[9.2.2 回收PV](#)

[9.2.3 PV动态供给](#)

[9.3 一个数据库例子](#)

[9.4 小结](#)

[第10章 Secret & Configmap](#)

[10.1 创建Secret](#)

[10.2 查看Secret](#)

[10.3 在Pod中使用Secret](#)

[10.3.1 Volume方式](#)

[10.3.2 环境变量方式](#)

[10.4 ConfigMap](#)

[10.5 小结](#)

[第11章 Helm—Kubernetes的包管理器](#)

[11.1 Why Helm](#)

[11.2 Helm架构](#)

[11.3 安装Helm](#)

[11.3.1 Helm客户端](#)

[11.3.2 Tiller服务器](#)

[11.4 使用Helm](#)

[11.5 chart详解](#)

[11.5.1 chart目录结构](#)

[11.5.2 chart模板](#)

[11.5.3 再次实践MySQL chart](#)

[11.5.4 升级和回滚release](#)

[11.5.5 开发自己的chart](#)

[11.6 小结](#)

[第12章 网络](#)

[12.1 Kubernetes网络模型](#)

[12.2 各种网络方案](#)

[12.3 Network Policy](#)

[12.3.1 部署Canal](#)

[12.3.2 实践Network Policy](#)

[12.4 小结](#)

[第13章 Kubernetes Dashboard](#)

[13.1 安装](#)

[13.2 配置登录权限](#)

[13.3 Dashboard界面结构](#)

[13.4 典型使用场景](#)

[13.4.1 部署Deployment](#)

[13.4.2 在线操作](#)

[13.4.3 查看资源详细信息](#)

[13.4.4 查看Pod日志](#)

[13.5 小结](#)

[第14章 Kubernetes集群监控](#)

[14.1 Weave Scope](#)

[14.1.1 安装Scope](#)

[14.1.2 使用Scope](#)

[14.2 Heapster](#)

[14.2.1 部署](#)

[14.2.2 使用](#)

[14.3 Prometheus Operator](#)

[14.3.1 Prometheus架构](#)

[14.3.2 Prometheus Operator架构](#)

[14.3.3 部署Prometheus Operator](#)

[14.4 小结](#)

[第15章 Kubernetes集群日志管理](#)

[15.1 部署](#)

[15.2 小结](#)

[写在最后](#)

本书封面贴有清华大学出版社防伪标签，无标签者不得销售

版权所有，侵权必究。侵权举报电话：010-62782989 13701121933

图书在版编目（CIP）数据

每天5分钟玩转Kubernetes / CloudMan著. — 北京：清华大学出版社，2018

ISBN 978-7-302-49667-0

I. ①每... II. ①C... III. ①Linux操作系统—程序设计 IV. ①TP316.85

中国版本图书馆CIP数据核字（2018）第033859号

责任编辑：夏毓彦

封面设计：王翔

责任校对：闫秀华

责任印制：王静怡

出版发行：清华大学出版社

网 址： <http://www.tup.com.cn>, <http://www.wqbook.com>

地 址：北京清华大学学研大厦A座

邮 编：100084

社总机：010-62770175

邮 购：010-62786544

投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn

质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn

印 刷 者：北京富博印刷有限公司

装 订 者：北京市密云县京文制本装订厂

经 销：全国新华书店

开 本：190mm×260mm

印 张：11.5

字 数：294千字

版 次：2018年4月第1版

印 次：2018年4月第1次印刷

印 数：1~3500

定 价：39.00元

产品编号：079113-01

内容简介

Kubernetes是容器编排引擎的事实标准，是继大数据、云计算和Docker之后又一热门技术，而且未来相当一段时间内都会非常流行。对于IT

行业来说，这是一项非常有价值的技术。对于IT从业者来说，掌握容器技术既是市场的需要，也是提升自我价值的重要途径。

本书共15章，系统介绍了Kubernetes的架构、重要概念、安装部署方法、运行管理应用的技术、网络存储管理、集群监控和日志管理等重要内容。书中通过大量实操案例深入浅出地讲解Kubernetes核心技术，是一本从入门到进阶的实用Kubernetes操作指导手册。读者在学习的过程中，可以跟着教程进行操作，在实践中掌握Kubernetes的核心技能。在之后的工作中，则可以将本教程作为参考书，按需查找相关知识点。

本书主要面向微服务软件开发人员，以及IT实施和运维工程师等相关人员，也适合作为高等院校和培训学校相关专业的教学参考书。

前言

写在最前面

《每天5分钟玩转Kubernetes》是一本系统学习Kubernetes的教程，有下面两个特点：

- 系统讲解当前最流行的容器编排引擎Kubernetes

包括安装部署、应用管理、网络、存储、监控、日志管理等多个方面。

- 重实践并兼顾理论

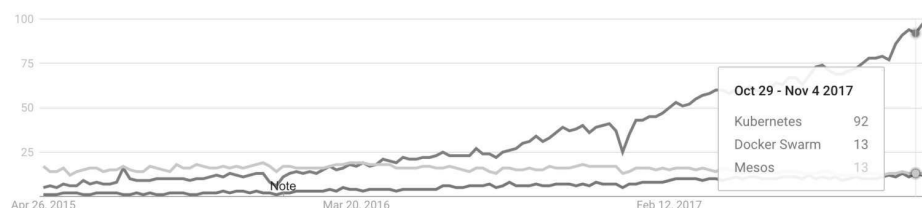
通过大量实验和操作带领大家学习Kubernetes。

为什么要写这个

因为Kubernetes非常热门，但学习门槛高。

2017年9月，Mesosphere宣布支持Kubernetes；10月，Docker宣布将在新版本中加入对Kubernetes的原生支持。至此，容器编排引擎领域的三足鼎立时代结束，Kubernetes赢得全面胜利。

其实早在2015年5月，Kubernetes在Google上的搜索热度就已经超过了Mesos和Docker Swarm，从那之后便是一路飙升，将对手“甩开了十几条街”。



目前，AWS、Azure、Google、阿里云、腾讯云等主流公有云提供的是基于Kubernetes的容器服务。Rancher、CoreOS、IBM、Mirantis、Oracle、Red Hat、VMWare等无数厂商也在大力研发和推广基于Kubernetes的容器CaaS或PaaS产品。可以说，Kubernetes是当前容器行业最热门的。

每一轮新技术的兴起，无论对公司还是个人既是机会也是挑战。这项新技术未来必将成为主流，那么作为IT从业者，正确的做法就是尽快掌握。因为：

(1) 新技术意味着新的市场和新的需求。初期掌握这种技术的人不是很多，而市场需求会越来越大，因而会形成供不应求的卖方市场，物以稀为贵，这对技术人员将是一个难得的价值提升机会。

(2) 学习新技术需要时间和精力，早起步早成材。

机会讲过了，咱们再来看看挑战。

新技术往往意味着技术上的突破和创新，会有不少新的概念和方法。

对于Kubernetes这项平台级技术，覆盖的技术范围非常广，包括计算、网络、存储、高可用、监控、日志管理等多个方面，要掌握这些新技术对IT老兵尚有不小难度，更别说新人了。

写给谁看

这套教程的目标读者包括：

IT实施和运维工程师

越来越多的应用将以容器的方式在开发、测试和生产环境中运行。掌握基于Kubernetes的容器平台运维能力将成为实施和运维工程师的核心竞争力。

软件开发人员

基于容器的微服务架构（Microservice Architecture）会逐渐成为开发应用系统的主流，而Kubernetes将是运行微服务应用的理想平台，市场将需要大量具备Kubernetes技能的应用程序开发人员。

我自己

CloudMan坚信最好的学习方法是分享。编写这本教程的同时也是对自己学习和实践Kubernetes技术的总结。对于知识，只有把它写出来并能够让其他人理解，才能说明自己真正掌握了。

著者

2018年1月

第1章 先把Kubernetes跑起来

Kubernetes（K8s）是Google在2014年发布的一个开源项目。

据说Google的数据中心里运行着20多亿个容器，而且Google十年前就开始使用容器技术。

最初，Google开发了一个叫Borg的系统（现在命名为Omega）来调度如此庞大数量的容器和工作负载。在积累了这么多年的经验后，Google决定重写这个容器管理系统，并将其贡献到开源社区，让全世界都能受益。

这个项目就是Kubernetes。简单地讲，Kubernetes是Google Omega的开源版本。

从2014年第一个版本发布以来，Kubernetes迅速获得开源社区的追捧，包括Red Hat、VMware、Canonical在内的很多有影响力的公司加入到开发和推广的阵营。目前Kubernetes已经成为发展最快、市场占有率最高的容器编排引擎产品。

Kubernetes一直在快速地开发和迭代。本书我们将以v1.7和v1.8为基础学习Kubernetes。我们会讨论Kubernetes重要的概念和架构，学习Kubernetes如何编排容器，包括优化资源利用、高可用、滚动更新、网络插件、服务发现、监控、数据管理、日志管理等。

下面就让我们开始Kubernetes的探险之旅。

1.1 先跑起来

按照一贯的学习思路，我们会在最短时间内搭建起一个可用系统，这样就能够尽快建立起对学习对象的感性认识。先把玩起来，快速了解基本概念、功能和使用场景。

越是门槛高的知识，就越需要有这么一个最小可用系统。如果直接上来就学习理论知识和概念，很容易从入门到放弃。

当然，要搭建这么一个可运行的系统通常也不会太容易，不过很幸运，Kubernetes官网已经为大家准备好了现成的最小可用系统。

kubernetes.io开发了一个交互式教程，通过Web浏览器就能使用预先部署好的一个Kubernetes集群，快速体验Kubernetes的功能和应用场景，下面我就带着大家去玩一下。

打开<https://kubernetes.io/docs/tutorials/kubernetes-basics/>。

页面左边就能看到教程菜单，如图1-1所示。

Tutorials

▼ Kubernetes Basics

Overview

- ▶ 1. Create a Cluster
- ▶ 2. Deploy an App
- ▶ 3. Explore Your App
- ▶ 4. Expose Your App Publicly
- ▶ 5. Scale Your App
- ▶ 6. Update Your App

图1-1

教程会指引大家完成创建Kubernetes集群、部署应用、访问应用、扩展应用、更新应用等最常见的使用场景，迅速建立感性认识。

1.2 创建Kubernetes集群

点击教程菜单 1. Create a Cluster → Interactive Tutorial - Creating a Cluster，如图1-2所示。

Tutorials

▼ Kubernetes Basics

Overview

▼ 1. Create a Cluster

Using Minikube to Create a Cluster

Interactive Tutorial - Creating a Cluster

- ▶ 2. Deploy an App
- ▶ 3. Explore Your App
- ▶ 4. Expose Your App Publicly
- ▶ 5. Scale Your App
- ▶ 6. Update Your App

图1-2

显示操作界面，如图1-3所示。

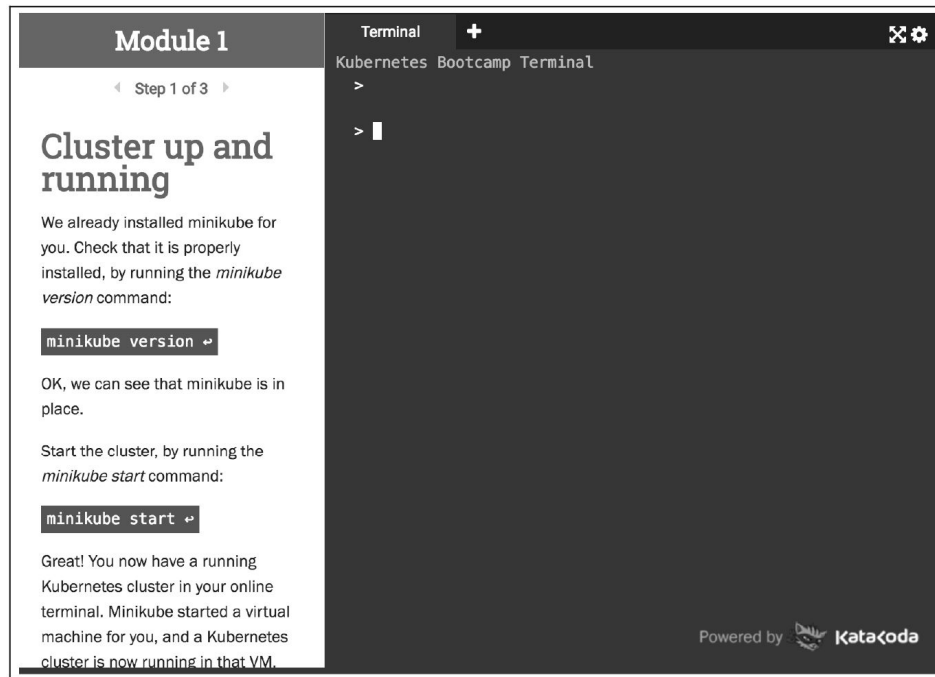
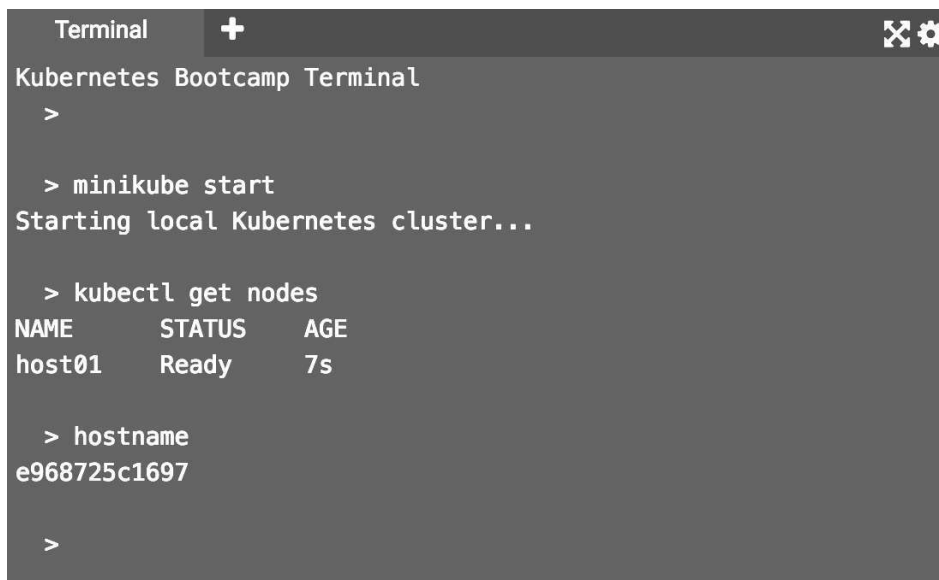


图1-3

左边部分是操作说明。右边是Terminal，即命令终端窗口。

按照操作说明，我们在Terminal中执行minikube start，然后执行kubectl get nodes，这样就创建好了一个单节点的kubernetes集群，如图1-4所示。



```
Terminal + [X] [G]
Kubernetes Bootcamp Terminal
>

> minikube start
Starting local Kubernetes cluster...

> kubectl get nodes
NAME      STATUS    AGE
host01    Ready     7s

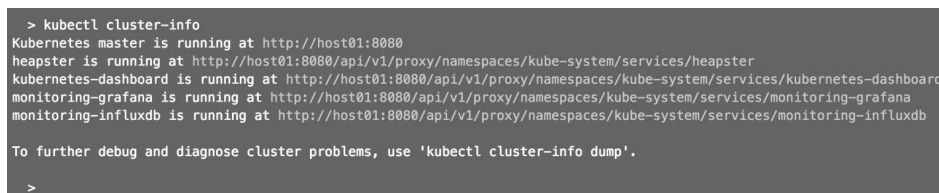
> hostname
e968725c1697

>
```

图1-4

集群的唯一节点为host01，需要注意的是当前执行命令的地方并不是host01。我们是通过Kubernetes的命令行工具kubectl远程管理集群。

执行kubectl cluster-info查看集群信息，如图1-5所示。



```
> kubectl cluster-info
Kubernetes master is running at http://host01:8080
heapster is running at http://host01:8080/api/v1/proxy/namespaces/kube-system/services/heapster
kubernetes-dashboard is running at http://host01:8080/api/v1/proxy/namespaces/kube-system/services/kubernetes-dashboard
monitoring-grafana is running at http://host01:8080/api/v1/proxy/namespaces/kube-system/services/monitoring-grafana
monitoring-influxdb is running at http://host01:8080/api/v1/proxy/namespaces/kube-system/services/monitoring-influxdb

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.

>
```

图1-5

heapster、kubernetes-dashboard都是集群中运行的服务。

注意：为节省篇幅，在后面的演示中，我们将简化操作步骤，详细的说明和完整步骤请参考官网在线文档。

1.3 部署应用

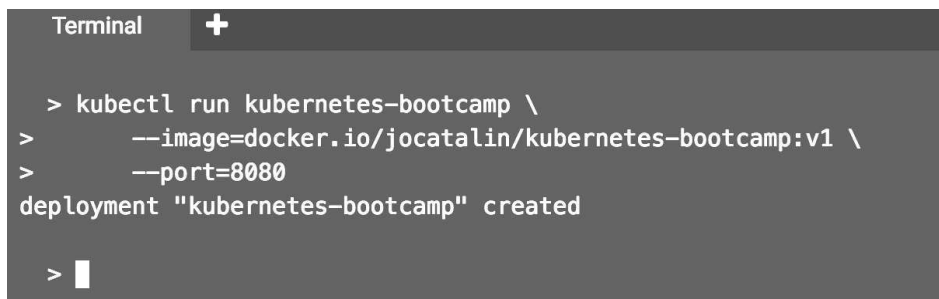
执行命令：

```
kubectl run kubernetes-bootcamp \  
  
  --image=docker.io/jocatalin/kubernetes-bootcamp:v1 \  
  
  --port=8080
```

这里我们通过 `kubectl run` 部署了一个应用，命名为 `kubernetes-bootcamp`，如图1-6所示。

Docker镜像通过`--image`指定。

`--port`设置应用对外服务的端口。



```
Terminal +  
  
> kubectl run kubernetes-bootcamp \  
>   --image=docker.io/jocatalin/kubernetes-bootcamp:v1 \  
>   --port=8080  
deployment "kubernetes-bootcamp" created  
  
> █
```

图1-6

这里Deployment是Kubernetes的术语，可以理解为应用。

Kubernetes还有一个重要术语Pod。

Pod是容器的集合，通常会将紧密相关的一组容器放到一个Pod中，同一个Pod中的所有容器共享IP地址和Port空间，也就是说它们在一个network namespace中。

Pod是Kubernetes调度的最小单位，同一Pod中的容器始终被一起调度。

运行`kubectl get pods`，查看当前的Pod，如图1-7所示。

```
> kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
kubernetes-bootcamp-390780338-q9p1t 1/1     Running   0           11m
>
```

图1-7

kubernetes-bootcamp-390780338-q9p1t就是应用的Pod。

1.4 访问应用

默认情况下，所有Pod只能在集群内部访问。对于上面这个例子，要访问应用只能直接访问容器的8080端口。为了能够从外部访问应用，我们需要将容器的8080端口映射到节点的端口。

执行如下命令，结果如图1-8所示。

```
kubectl expose deployment/kubernetes-bootcamp \

--type="NodePort" \

--port 8080
```

```
> kubectl expose deployment/kubernetes-bootcamp \
>     --type="NodePort" \
>     --port 8080
service "kubernetes-bootcamp" exposed
>
```

图1-8

执行命令kubectl get services，可以查看应用被映射到节点的哪个端口，如图1-9所示。

```
> kubectl get services
NAME                CLUSTER-IP    EXTERNAL-IP    PORT(S)          AGE
kubernetes           10.0.0.1      <none>         443/TCP          2m
kubernetes-bootcamp  10.0.0.131    <nodes>        8080:32320/TCP   1m
```

图1-9

这里有两个service，可以将service暂时理解为端口映射，后面我们会详细讨论。

Kubernetes是默认的service，暂时不用考虑。kubernetes-bootcamp是我们应用的service，8080端口已经映射到host01的32320端口，端口号是随机分配的，可以执行如下命令访问应用，结果如图1-10所示。

```
curl host01:32320
```

```
> curl host01:32320
Hello Kubernetes bootcamp! | Running on: kubernetes-bootcamp-390780338-q9p1t | v=1
>
```

图1-10

1.5 Scale应用

默认情况下应用只会运行一个副本，可以通过kubectl get deployments查看副本数，如图1-11所示。

```
> kubectl get deployments
NAME                DESIRED    CURRENT    UP-TO-DATE    AVAILABLE    AGE
kubernetes-bootcamp  1          1          1             1            14m
```

图1-11

执行如下命令将副本数增加到3个，如图1-12所示。

```
kubectl scale deployments/kubernetes-bootcamp --replicas=3
```

```

> kubectl scale deployments/kubernetes-bootcamp --replicas=3
deployment "kubernetes-bootcamp" scaled

>

> kubectl get deployments
NAME                DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
kubernetes-bootcamp 3          3         3            3           17m

```

图1-12

通过kubectl get pods可以看到当前Pod增加到3个，如图1-13所示。

```

> kubectl get pods
NAME                                     READY   STATUS    RESTARTS   AGE
kubernetes-bootcamp-390780338-12sbg     1/1     Running   0          1m
kubernetes-bootcamp-390780338-q9p1t     1/1     Running   0          19m
kubernetes-bootcamp-390780338-swvp7     1/1     Running   0          1m

```

图1-13

通过curl访问应用，可以看到每次请求发送到不同的Pod，3个副本轮询处理，这样就实现了负载均衡，如图1-14所示。

```

> curl host01:32320
Hello Kubernetes bootcamp! | Running on: kubernetes-bootcamp-390780338-12sbg | v=1

> curl host01:32320
Hello Kubernetes bootcamp! | Running on: kubernetes-bootcamp-390780338-swvp7 | v=1

> curl host01:32320
Hello Kubernetes bootcamp! | Running on: kubernetes-bootcamp-390780338-q9p1t | v=1

> curl host01:32320
Hello Kubernetes bootcamp! | Running on: kubernetes-bootcamp-390780338-12sbg | v=1

>

```

图1-14

要scale down也很方便，执行下列命令，结果如图1-15所示。

```
kubectl scale deployments/kubernetes-bootcamp --replicas=2
```

```

> kubectl scale deployments/kubernetes-bootcamp --replicas=2
deployment "kubernetes-bootcamp" scaled

> kubectl get deployments
NAME                DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
kubernetes-bootcamp 2          2         2            2           25m

> kubectl get pods
NAME                                     READY   STATUS    RESTARTS   AGE
kubernetes-bootcamp-390780338-12sbg     1/1     Running   0          8m
kubernetes-bootcamp-390780338-q9p1t     1/1     Running   0          25m
kubernetes-bootcamp-390780338-swp7      1/1     Terminating 0          8m

```

图1-15

从图1-15中可以看到，其中一个副本被删除了。

1.6 滚动更新

当前应用使用的image版本为v1，执行如下命令将其升级到v2，结果如图1-16所示。

```
kubectl set image deployments/kubernetes-bootcamp
```

```
kubernetes-bootcamp=jocatalin/kubernetes-bootcamp:v2
```

```

> kubectl set image deployments/kubernetes-bootcamp kubernetes-bootcamp=jocatalin/kubernetes-bootcamp:v2
deployment "kubernetes-bootcamp" image updated

> kubectl get pods
NAME                                     READY   STATUS    RESTARTS   AGE
kubernetes-bootcamp-2100875782-2q5k8    0/1     ContainerCreating 0          6s
kubernetes-bootcamp-2100875782-sbwkc     0/1     ContainerCreating 0          4s
kubernetes-bootcamp-390780338-12sbg     1/1     Terminating 0          19m
kubernetes-bootcamp-390780338-q9p1t     1/1     Running       0          36m

> kubectl get pods
NAME                                     READY   STATUS    RESTARTS   AGE
kubernetes-bootcamp-2100875782-2q5k8    1/1     Running    0          15s
kubernetes-bootcamp-2100875782-sbwkc     1/1     Running    0          13s
kubernetes-bootcamp-390780338-12sbg     1/1     Terminating 0          19m
kubernetes-bootcamp-390780338-q9p1t     1/1     Terminating 0          37m

> kubectl get pods
NAME                                     READY   STATUS    RESTARTS   AGE
kubernetes-bootcamp-2100875782-2q5k8    1/1     Running    0          1m
kubernetes-bootcamp-2100875782-sbwkc     1/1     Running    0          1m
>

```

图1-16

通过`kubectl get pods`可以观察滚动更新的过程：`v1`的Pod被逐个删除，同时启动了新的`v2` Pod。更新完成后访问新版本应用，如图1-17所示。

```
> curl host01:32320
Hello Kubernetes bootcamp! | Running on: kubernetes-bootcamp-2100875782-sbwkc | v=2

> curl host01:32320
Hello Kubernetes bootcamp! | Running on: kubernetes-bootcamp-2100875782-2q5k8 | v=2

>
```

图1-17

如果要回退到`v1`版本也很容易，执行`kubectl rollout undo`命令，结果如图1-18所示。

`kubectl rollout undo deployments/kubernetes-bootcamp`

```
> kubectl rollout undo deployments/kubernetes-bootcamp
deployment "kubernetes-bootcamp" rolled back

> kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
kubernetes-bootcamp-2100875782-2q5k8	1/1	Running	0	7m
kubernetes-bootcamp-2100875782-sbwkc	1/1	Terminating	0	7m
kubernetes-bootcamp-390780338-9kvjj	0/1	ContainerCreating	0	5s
kubernetes-bootcamp-390780338-hmcgb	0/1	ContainerCreating	0	5s

图1-18

验证版本已经回退到`v1`，如图1-19所示。

```
> curl host01:32320
Hello Kubernetes bootcamp! | Running on: kubernetes-bootcamp-390780338-hmcgb | v=1

> curl host01:32320
Hello Kubernetes bootcamp! | Running on: kubernetes-bootcamp-390780338-9kvjj | v=1

>
```

图1-19

1.7 小结

至此，我们已经通过官网的交互式教程快速体验了Kubernetes的功能和使用方法。本书的其余章节将详细讨论Kubernetes的架构、典型的部署方法、容器编排能力、网络方案、监控方案，帮助大家全面掌握Kubernetes的核心技能。

第2章 重要概念

在实践之前，必须先学习Kubernetes的几个重要概念，它们是组成Kubernetes集群的基石。

1. Cluster

Cluster是计算、存储和网络资源的集合，Kubernetes利用这些资源运行各种基于容器的应用。

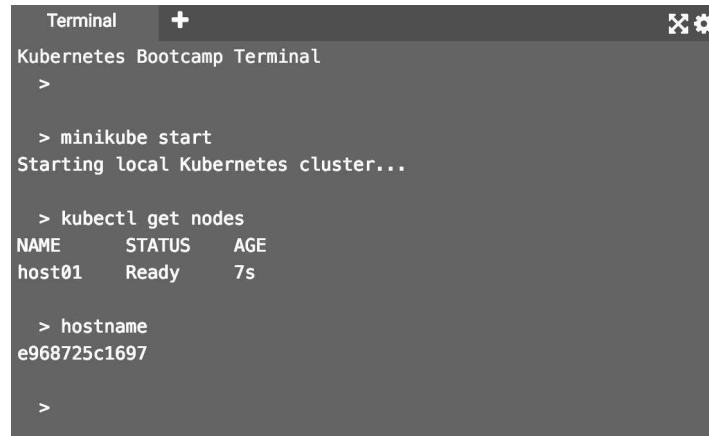
2. Master

Master是Cluster的大脑，它的主要职责是调度，即决定将应用放在哪里运行。Master运行Linux操作系统，可以是物理机或者虚拟机。为了实现高可用，可以运行多个Master。

3. Node

Node的职责是运行容器应用。Node由Master管理，Node负责监控并汇报容器的状态，同时根据Master的要求管理容器的生命周期。Node运行在Linux操作系统上，可以是物理机或者是虚拟机。

在前面交互式教程中，我们创建的Cluster只有一个主机host01，它既是Master也是Node，如图2-1所示。

A terminal window titled "Terminal" with a plus icon and a settings icon. The window shows the output of several commands in a Kubernetes Bootcamp environment. The commands and their outputs are:
1. `>` (prompt)
2. `> minikube start` followed by `Starting local Kubernetes cluster...`
3. `> kubectl get nodes` followed by a table:

NAME	STATUS	AGE
host01	Ready	7s

4. `> hostname` followed by `e968725c1697`
5. `>` (prompt)

```
Terminal +
Kubernetes Bootcamp Terminal
>

> minikube start
Starting local Kubernetes cluster...

> kubectl get nodes
NAME      STATUS    AGE
host01    Ready     7s

> hostname
e968725c1697

>
```

图2-1

4. Pod

Pod是Kubernetes的最小工作单元。每个Pod包含一个或多个容器。Pod中的容器会作为一个整体被Master调度到一个Node上运行。

Kubernetes引入Pod主要基于下面两个目的：

(1) 可管理性。

有些容器天生就是需要紧密联系，一起工作。Pod提供了比容器更高层次的抽象，将它们封装到一个部署单元中。Kubernetes以Pod为最小单位进行调度、扩展、共享资源、管理生命周期。

(2) 通信和资源共享。

Pod中的所有容器使用同一个网络namespace，即相同的IP地址和Port空间。它们可以直接用localhost通信。同样的，这些容器可以共享存储，当Kubernetes挂载volume到Pod，本质上是将volume挂载到Pod中的每一个容器。

Pods有两种使用方式：

(1) 运行单一容器。

one-container-per-Pod是Kubernetes最常见的模型，这种情况下，只是将单个容器简单封装成Pod。即便是只有一个容器，Kubernetes管理的也

是Pod而不是直接管理容器。

(2) 运行多个容器。

问题在于：哪些容器应该放到一个Pod中？

答案是：这些容器联系必须非常紧密，而且需要直接共享资源。

举个例子，如图2-2所示，这个Pod包含两个容器：一个是File Puller，一个是Web Server。

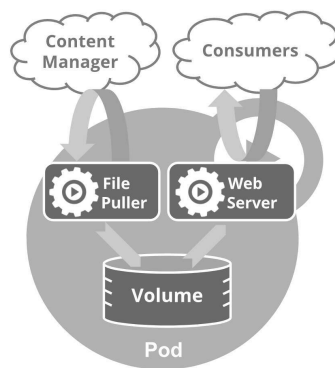


图2-2

File Puller会定期从外部的Content Manager中拉取最新的文件，将其存放在共享的volume中。Web Server从volume读取文件，响应Consumer的请求。

这两个容器是紧密协作的，它们一起为Consumer提供最新的数据；同时它们也通过volume共享数据，所以放到一个Pod是合适的。

再来看一个反例：是否需要将Tomcat和MySQL放到一个Pod中？

Tomcat从MySQL读取数据，它们之间需要协作，但还不至于需要放到一个Pod中一起部署、一起启动、一起停止。同时它们之间是通过JDBC交换数据，并不是直接共享存储，所以放到各自的Pod中更合适。

5. Controller

Kubernetes通常不会直接创建Pod，而是通过Controller来管理Pod的。Controller中定义了Pod的部署特性，比如有几个副本、在什么样的Node上运行等。为了满足不同的业务场景，Kubernetes提供了多种Controller，包括Deployment、ReplicaSet、DaemonSet、StatefulSet、Job等，我们逐一讨论。

（1）Deployment是最常用的Controller，比如在线教程中就是通过创建Deployment来部署应用的。Deployment可以管理Pod的多个副本，并确保Pod按照期望的状态运行。

（2）ReplicaSet实现了Pod的多副本管理。使用Deployment时会自动创建ReplicaSet，也就是说Deployment是通过ReplicaSet来管理Pod的多个副本的，我们通常不需要直接使用ReplicaSet。

（3）DaemonSet用于每个Node最多只运行一个Pod副本的场景。正如其名称所揭示的，DaemonSet通常用于运行daemon。

（4）StatefulSet能够保证Pod的每个副本在整个生命周期中名称是不变的，而其他Controller不提供这个功能。当某个Pod发生故障需要删除并重新启动时，Pod的名称会发生变化，同时StatefulSet会保证副本按照固定的顺序启动、更新或者删除。

（5）Job用于运行结束就删除的应用，而其他Controller中的Pod通常是长期持续运行。

6. Service

Deployment可以部署多个副本，每个Pod都有自己的IP，外界如何访问这些副本呢？

通过Pod的IP吗？

要知道Pod很可能会被频繁地销毁和重启，它们的IP会发生变化，用IP来访问不太现实。

答案是Service。

Kubernetes Service定义了外界访问一组特定Pod的方式。Service有自己的IP和端口，Service为Pod提供了负载均衡。

Kubernetes运行容器（Pod）与访问容器（Pod）这两项任务分别由Controller和Service执行。

7. Namespace

如果有多个用户或项目组使用同一个Kubernetes Cluster，如何将他们创建的Controller、Pod等资源分开呢？

答案就是Namespace。

Namespace可以将一个物理的Cluster逻辑上划分成多个虚拟Cluster，每个Cluster就是一个Namespace。不同Namespace里的资源是完全隔离的。

Kubernetes默认创建了两个Namespace，如图2-3所示。

```
> kubectl get namespace
NAME          STATUS    AGE
default       Active    17s
kube-system   Active    17s
>
```

图2-3

- default: 创建资源时如果不指定，将被放到这个Namespace中。
- kube-system: Kubernetes自己创建的系统资源将放到这个Namespace中。

第3章 部署Kubernetes Cluster

本章我们将部署三个节点的Kubernetes Cluster，如图3-1所示。

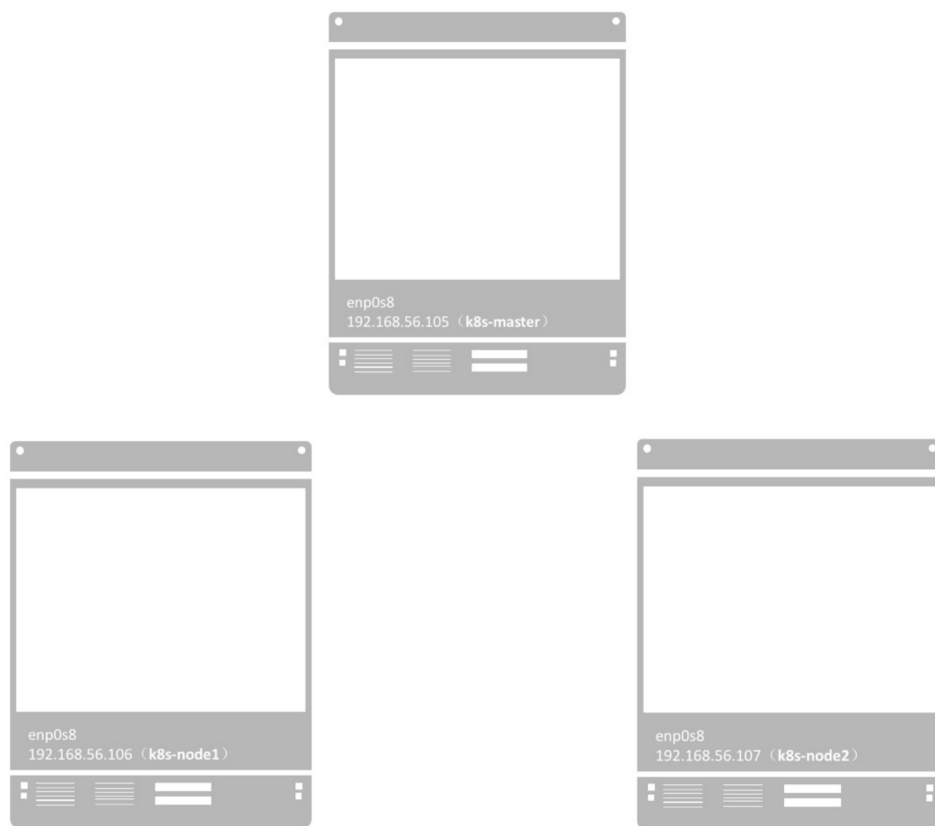


图3-1

k8s-master是Master，k8s-node1和k8s-node2是Node。

所有节点的操作系统均为Ubuntu 16.04，当然其他Linux也是可以的。

官方安装文档可以参考 <https://kubernetes.io/docs/setup/independent/install-kubeadm/>。

注意：Kubernetes几乎所有的安装组件和Docker镜像都放在Google自己的网站上，这对国内的同学可能是个不小的障碍。建议是：网络障碍都必须想办法克服，不然连Kubernetes的门都进不了。

3.1 安装Docker

所有节点都需要安装Docker。

```
apt-get update && apt-get install docker.io
```

3.2 安装kubectlet、kubeadm和kubectl

在所有节点上安装kubectlet、kubeadm和kubectl。

- kubectlet运行在Cluster所有节点上，负责启动Pod和容器。
- kubeadm用于初始化Cluster。
- kubectl是Kubernetes命令行工具。通过kubectl可以部署和管理应用，查看各种资源，创建、删除和更新各种组件。

```
apt-get update && apt-get install -y apt-transport-https
```

```
curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | apt-key add -
```

```
cat <<EOF >/etc/apt/sources.list.d/kubernetes.list
```

```
deb http://apt.kubernetes.io/ kubernetes-xenial main
```

```
EOF
```

```
apt-get update
```

```
apt-get install -y kubectlet kubeadm kubectl
```

3.3 用kubeadm创建Cluster

完整的官方文档可以参考 <https://kubernetes.io/docs/setup/independent/create-cluster-kubeadm/>。

3.3.1 初始化Master

在Master上执行如下命令：

```
kubeadm init --apiserver-advertise-address 192.168.56.105
```

```
--pod-network-cidr=10.244.0.0/16
```

`--apiserver-advertise-address`指明用Master的哪个interface与Cluster的其他节点通信。如果Master有多个interface，建议明确指定，如果不指定，`kubeadm`会自动选择有默认网关的interface。

`--pod-network-cidr`指定Pod网络的范围。Kubernetes支持多种网络方案，而且不同网络方案对`--pod-network-cidr`有自己的要求，这里设置为`10.244.0.0/16`是因为我们将使用flannel网络方案，必须设置成这个CIDR。在后面的实践中我们会切换到其他网络方案，比如Canal。

初始化过程如图3-2所示。

```
root@k8s-master:~#
root@k8s-master:~# kubeadm init --apiserver-advertise-address 192.168.56.105 --pod-network-cidr=10.244.0.0/16
[kubeadm] WARNING: kubeadm is in beta, please do not use it for production clusters.
[init] Using Kubernetes version: v1.7.4
[init] Using Authorization modes: [Node RBAC]
[preflight] Running pre-flight checks ①
[preflight] Starting the kubelet service
[kubeadm] WARNING: starting in 1.8, tokens expire after 24 hours by default (if you require a non-expiring token use --token-ttl 0)
[certificates] Generated CA certificate and key.
[certificates] Generated API server certificate and key. ②
[certificates] API Server serving cert is signed for DNS names [k8s-master kubernetes kubernetes.default kubernetes.default.svc kubernetes.default.svc.cluster.local]
[certificates] Generated API server kubelet client certificate and key.
[certificates] Generated service account token signing key and public key.
[certificates] Generated front-proxy CA certificate and key.
[certificates] Generated front-proxy client certificate and key.
[certificates] Valid certificates and keys now exist in "/etc/kubernetes/pki"
[kubeconfig] Wrote KubeConfig file to disk: "/etc/kubernetes/admin.conf"
[kubeconfig] Wrote KubeConfig file to disk: "/etc/kubernetes/kubelet.conf" ③
[kubeconfig] Wrote KubeConfig file to disk: "/etc/kubernetes/controller-manager.conf"
[kubeconfig] Wrote KubeConfig file to disk: "/etc/kubernetes/scheduler.conf"
[apiclient] Created API client, waiting for the control plane to become ready ④
[apiclient] All control plane components are healthy after 26.505992 seconds
[token] Using token: d38a01.13653e584ccc1980
[apiconfig] Created RBAC rules
[addons] Applied essential addon: kube-proxy ⑤
[addons] Applied essential addon: kube-dns

Your Kubernetes master has initialized successfully! ⑥

To start using your cluster, you need to run (as a regular user):

mkdir -p $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config ⑦
sudo chown $(id -u):$(id -g) $HOME/.kube/config

You should now deploy a pod network to the cluster.
Run "kubectl apply -f [podnetwork].yaml" with one of the options listed at: ⑧
http://kubernetes.io/docs/admin/addons/

You can now join any number of machines by running the following on each node
as root:

kubeadm join --token d38a01.13653e584ccc1980 192.168.56.105:6443 ⑨

root@k8s-master:~#
```

图3-2

- (1) `kubeadm`执行初始化前的检查。
- (2) 生成token和证书。
- (3) 生成KubeConfig文件，`kubelet`需要用这个文件与Master通信。
- (4) 安装Master组件，会从Google的Registry下载组件的Docker镜像。这一步可能会花一些时间，主要取决于网络质量。

- (5) 安装附加组件kube-proxy和kube-dns。
- (6) Kubernetes Master初始化成功。
- (7) 提示如何配置kubectl，后面会实践。
- (8) 提示如何安装Pod网络，后面会实践。
- (9) 提示如何注册其他节点到Cluster，后面会实践。

3.3.2 配置kubectl

kubectl是管理Kubernetes Cluster的命令行工具，前面我们已经在所有的节点安装了kubectl。Master初始化完成后需要做一些配置工作，然后kubectl就能使用了。

依照kubeadm init输出的第7步提示，推荐用Linux普通用户执行kubectl（root会有一些问题）。

我们为用户ubuntu配置kubectl：

```
su - ubuntu
```

```
mkdir -p $HOME/.kube
```

```
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
```

```
sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

为了使用更便捷，启用kubectl命令的自动补全功能：

```
echo "source <(kubectl completion bash)" >> ~/.bashrc
```

这样，用户ubuntu就可以使用kubectl了。

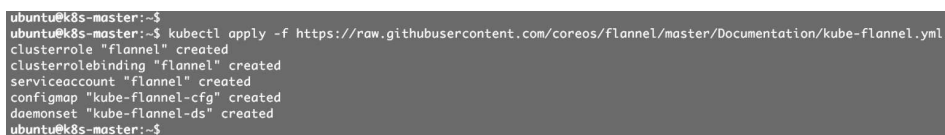
3.3.3 安装Pod网络

要让Kubernetes Cluster能够工作，必须安装Pod网络，否则Pod之间无法通信。

Kubernetes支持多种网络方案，这里我们先使用flannel，后面还会讨论Canal。

执行如下命令部署flannel，如图3-3所示。

```
kubectl apply -f https://raw.githubusercontent.com/coreos/flannel/master/Documentation/kube-flannel.yml
```



```
ubuntu@k8s-master:~$ kubectl apply -f https://raw.githubusercontent.com/coreos/flannel/master/Documentation/kube-flannel.yml
clusterrole "flannel" created
clusterrolebinding "flannel" created
serviceaccount "flannel" created
configmap "kube-flannel-cfg" created
daemonset "kube-flannel-ds" created
ubuntu@k8s-master:~$
```

图3-3

3.3.4 添加k8s-node1和k8s-node2

在k8s-node1和k8s-node2上分别执行如下命令，将其注册到Cluster中：

```
kubeadm join --token d38a01.13653e584ccc1980 192.168.56.105:6443
```

这里的--token来自前面kubeadm init输出的第9步提示，如果当时没有记录下来，可以通过kubeadm token list查看，如图3-4所示。



```
root@k8s-master:~# kubeadm token list
TOKEN          TTL    EXPIRES    USAGES          DESCRIPTION
d38a01.13653e584ccc1980  <forever>  <never>  authentication,signing  The default bootstrap token generated by 'kubeadm init'.
```

图3-4

kubeadm join执行如图3-5所示。

```

root@k8s-node1:~#
root@k8s-node1:~# kubeadm join --token d38a01.13653e584ccc1980 192.168.56.105:6443
[kubeadm] WARNING: kubeadm is in beta, please do not use it for production clusters.
[preflight] Running pre-flight checks
[preflight] Starting the kubelet service
[discovery] Trying to connect to API Server "192.168.56.105:6443"
[discovery] Created cluster-info discovery client, requesting info from "https://192.168.56.105:6443"
[discovery] Cluster info signature and contents are valid, will use API Server "https://192.168.56.105:6443"
[discovery] Successfully established connection with API Server "192.168.56.105:6443"
[bootstrap] Detected server version: v1.7.4
[bootstrap] The server supports the Certificates API (certificates.k8s.io/v1beta1)
[csr] Created API client to obtain unique certificate for this node, generating keys and certificate signing request
[csr] Received signed certificate from the API server, generating KubeConfig...
[kubeconfig] Wrote KubeConfig file to disk: "/etc/kubernetes/kubelet.conf"

Node join complete:
* Certificate signing request sent to master and response
  received.
* Kubelet informed of new secure connection details.

Run 'kubectl get nodes' on the master to see this machine join.
root@k8s-node1:~#

```

图3-5

根据提示，我们可以通过**kubectl get nodes**查看节点的状态，如图3-6所示。

```

ubuntu@k8s-master:~$ kubectl get nodes

```

NAME	STATUS	AGE	VERSION
k8s-master	NotReady	45m	v1.7.4
k8s-node1	NotReady	59s	v1.7.4
k8s-node2	NotReady	6s	v1.7.4

图3-6

目前所有节点都是**NotReady**，这是因为每个节点都需要启动若干组件，这些组件都是在**Pod**中运行，需要首先从**Google**下载镜像。我们可以通过如下命令查看**Pod**的状态，如图3-7所示。

kubectl get pod --all-namespaces

```

ubuntu@k8s-master:~$ kubectl get pod --all-namespaces

```

NAMESPACE	NAME	READY	STATUS	RESTARTS	AGE
kube-system	etcd-k8s-master	1/1	Running	0	44m
kube-system	kube-apiserver-k8s-master	1/1	Running	0	44m
kube-system	kube-controller-manager-k8s-master	1/1	Running	0	44m
kube-system	kube-dns-2425271678-1z3pv	0/3	Pending	0	49m
kube-system	kube-flannel-ds-cqbpb	0/2	ContainerCreating	0	6m
kube-system	kube-flannel-ds-v0p3x	0/2	ImagePullBackOff	0	9m
kube-system	kube-flannel-ds-xk49w	0/2	ContainerCreating	0	5m
kube-system	kube-proxy-16mg9	0/1	ContainerCreating	0	6m
kube-system	kube-proxy-wc4j0	1/1	Running	0	49m
kube-system	kube-proxy-xl5gd	0/1	ContainerCreating	0	5m
kube-system	kube-scheduler-k8s-master	1/1	Running	0	44m

```

ubuntu@k8s-master:~$

```

图3-7

Pending、ContainerCreating、ImagePullBackOff都表明Pod没有就绪，Running才是就绪状态。我们可以通过kubectl describe pod <Pod Name>查看Pod的具体情况，比如：

```
kubectl describe pod kube-flannel-ds-v0p3x --namespace=kube-system
```

结果如图3-8所示。

From	SubjectPath	Type	Reason	Message
kubernetes-master		Normal	SuccessfulMountVolume	MountVolume.Setup succeeded for volume "cni"
kubernetes-master		Normal	SuccessfulMountVolume	MountVolume.Setup succeeded for volume "run"
kubernetes-master		Normal	SuccessfulMountVolume	MountVolume.Setup succeeded for volume "flannel-cfg"
kubernetes-master		Normal	SuccessfulMountVolume	MountVolume.Setup succeeded for volume "flannel-tls-cfg"
kubernetes-master	spec.containers[kube-flannel]	Warning	Failed	Failed to pull image "quay.io/coreos/flannel:v0.8.0-amd64": rpc error: code = Unknown desc = Error: image quay.io/coreos/flannel:v0.8.0-amd64 not found
kubernetes-master	spec.containers[install-cni]	Normal	BackOff	Back-off pulling image "quay.io/coreos/flannel:v0.8.0-amd64"
kubernetes-master	spec.containers[kube-flannel]	Warning	FailedSync	Error syncing pod
kubernetes-master	spec.containers[kube-flannel]	Normal	BackOff	Back-off pulling image "quay.io/coreos/flannel:v0.8.0-amd64"
kubernetes-master	spec.containers[kube-flannel]	Normal	Pulling	Pulling image "quay.io/coreos/flannel:v0.8.0-amd64"

图3-8

为了节省篇幅，这里只截取命令输出的最后部分，可以看到在下载 `image` 时失败，如果网络质量不好，这种情况是很常见的。我们可以耐心等待，因为 `Kubernetes` 会重试，我们也可以自己手动执行 `docker pull` 去下载这个镜像。

等待一段时间，**image**成功下载后，所有**Pod**都会处于**Running**状态，如图3-9所示。

```
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ kubectl get pod --all-namespaces  
NAMESPACE      NAME                                                    READY    STATUS    RESTARTS   AGE  
kube-system     etcd-k8s-master                                         1/1      Running   0           10m  
kube-system     kube-apiserver-k8s-master                             1/1      Running   0           10m  
kube-system     kube-controller-manager-k8s-master                   1/1      Running   0           10m  
kube-system     kube-dns-2425271678-1z3pv                             3/3      Running   0           1h  
kube-system     kube-flannel-ds-cqbpb                                  2/2      Running   4           18m  
kube-system     kube-flannel-ds-v0p3x                                  2/2      Running   0           21m  
kube-system     kube-flannel-ds-xk49w                                  2/2      Running   0           17m  
kube-system     kube-proxy-16mg9                                        1/1      Running   0           18m  
kube-system     kube-proxy-wc4j0                                        1/1      Running   0           1h  
kube-system     kube-proxy-xl5gd                                        1/1      Running   0           17m  
kube-system     kube-scheduler-k8s-master                             1/1      Running   0           10m  
ubuntu@k8s-master:~$
```

图3-9

这时，所有的节点都已经准备好了，Kubernetes Cluster创建成功，如图3-10所示。

```
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ kubectl get nodes  
NAME          STATUS    AGE      VERSION  
k8s-master    Ready     1h       v1.7.4  
k8s-node1     Ready     18m      v1.7.4  
k8s-node2     Ready     17m      v1.7.4  
ubuntu@k8s-master:~$
```

图3-10

3.4 小结

本章通过kubeadm部署了三节点的Kubernetes集群，后面章节我们将在这个实验环境中学习Kubernetes的各项技术。

第4章 Kubernetes架构

Kubernetes Cluster由Master和Node组成，节点上运行着若干Kubernetes服务。

4.1 Master节点

Master是Kubernetes Cluster的大脑，运行着的Daemon服务包括kube-apiserver、kube-scheduler、kube-controller-manager、etcd和Pod网络（例如flannel），如图4-1所示。

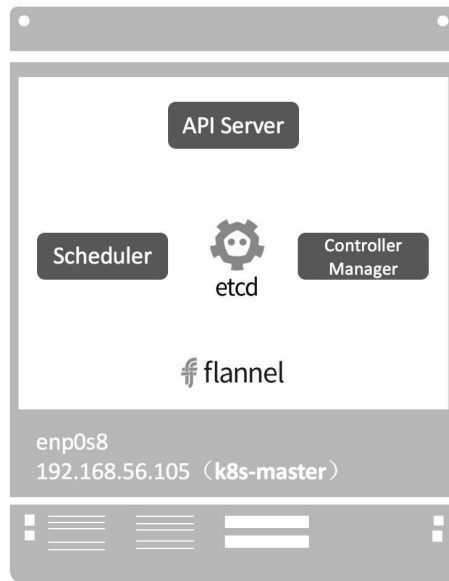


图4-1

1. API Server (kube-apiserver)

API Server 提供 HTTP/HTTPS RESTful API，即 Kubernetes API。API Server 是 Kubernetes Cluster 的前端接口，各种客户端工具（CLI 或 UI）以及 Kubernetes 其他组件可以通过它管理 Cluster 的各种资源。

2. Scheduler (kube-scheduler)

Scheduler 负责决定将 Pod 放在哪个 Node 上运行。Scheduler 在调度时会充分考虑 Cluster 的拓扑结构，当前各个节点的负载，以及应用对高可用、性能、数据亲和性的需求。

3. Controller Manager (kube-controller-manager)

Controller Manager 负责管理 Cluster 各种资源，保证资源处于预期的状态。Controller Manager 由多种 controller 组成，包括 replication controller、endpoints controller、namespace controller、serviceaccounts controller 等。

不同的 controller 管理不同的资源。例如，replication controller 管理 Deployment、StatefulSet、DaemonSet 的生命周期，namespace controller 管理 Namespace 资源。

4. etcd

etcd负责保存Kubernetes Cluster的配置信息和各种资源的状态信息。当数据发生变化时，etcd会快速地通知Kubernetes相关组件。

5. Pod网络

Pod要能够相互通信，Kubernetes Cluster必须部署Pod网络，flannel是其中一个可选方案。

以上是Master上运行的组件，下面我们接着讨论Node。

4.2 Node节点

Node是Pod运行的地方，Kubernetes支持Docker、rkt等容器Runtime。Node上运行的Kubernetes组件有kubelet、kube-proxy和Pod网络（例如flannel），如图4-2所示。

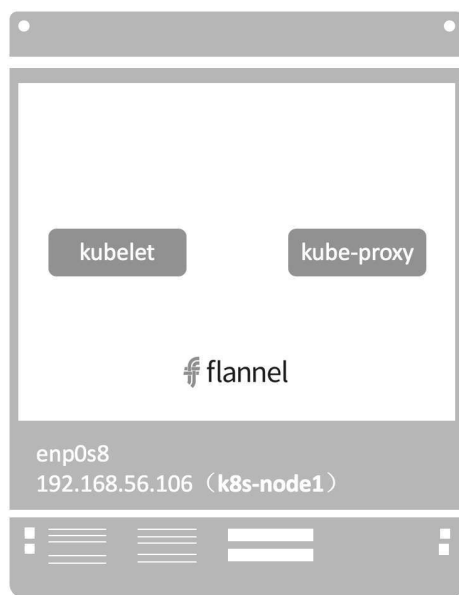


图4-2

1. kubelet

kubelet是Node的agent，当Scheduler确定在某个Node上运行Pod后，会将Pod的具体配置信息（image、volume等）发送给该节点的kubelet，kubelet根据这些信息创建和运行容器，并向Master报告运行状态。

2. kube-proxy

service在逻辑上代表了后端的多个Pod，外界通过service访问Pod。service接收到的请求是如何转发到Pod的呢？这就是kube-proxy要完成的工作。

每个Node都会运行kube-proxy服务，它负责将访问service的TCP/UDP数据流转发到后端的容器。如果有多个副本，kube-proxy会实现负载均衡。

3. Pod网络

Pod要能够相互通信，Kubernetes Cluster必须部署Pod网络，flannel是其中一个可选方案。

4.3 完整的架构图

结合实验环境，我们得到了如图4-3所示的架构图。

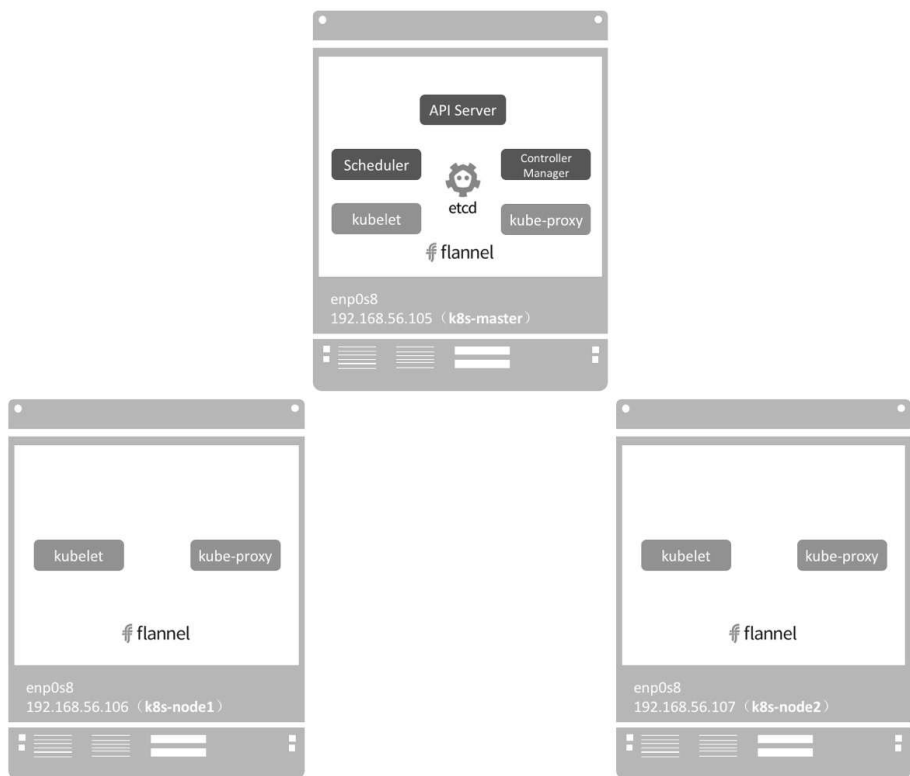


图4-3

你可能会问：为什么k8s-master上也有kubelet和kube-proxy呢？

这是因为Master上也可以运行应用，即Master同时也是一个Node。

几乎所有的Kubernetes组件本身也运行在Pod里，执行如下命令，结果如图4-4所示。

`kubectl get pod --all-namespaces -o wide`

```
ubuntu@k8s-master:~$ kubectl get pod --all-namespaces -o wide
```

NAMESPACE	NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
kube-system	etcd-k8s-master	1/1	Running	0	1d	192.168.56.105	k8s-master
kube-system	kube-apiserver-k8s-master	1/1	Running	0	1d	192.168.56.105	k8s-master
kube-system	kube-controller-manager-k8s-master	1/1	Running	0	1d	192.168.56.105	k8s-master
kube-system	kube-dns-2425271678-1z3pv	3/3	Running	0	1d	10.244.1.57	k8s-node1
kube-system	kube-flannel-ds-cqbpb	2/2	Running	4	1d	192.168.56.106	k8s-node1
kube-system	kube-flannel-ds-v0p3x	2/2	Running	0	1d	192.168.56.105	k8s-master
kube-system	kube-flannel-ds-xk49w	2/2	Running	0	1d	192.168.56.107	k8s-node2
kube-system	kube-proxy-16mg9	1/1	Running	0	1d	192.168.56.106	k8s-node1
kube-system	kube-proxy-wc4j0	1/1	Running	0	1d	192.168.56.105	k8s-master
kube-system	kube-proxy-xl5gd	1/1	Running	0	1d	192.168.56.107	k8s-node2
kube-system	kube-scheduler-k8s-master	1/1	Running	0	1d	192.168.56.105	k8s-master

```
ubuntu@k8s-master:~$
```

图4-4

Kubernetes的系统组件都被放到kube-system namespace中。这里有一个kube-dns组件，它为Cluster提供DNS服务，我们后面会讨论到。kube-dns是在执行kubeadm init时（第5步）作为附加组件安装的。

kubelet是唯一没有以容器形式运行的Kubernetes组件，它在Ubuntu中通过Systemd服务运行，如图4-5所示。

```
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ sudo systemctl status kubelet.service  
• kubelet.service - kubelet: The Kubernetes Node Agent  
Loaded: loaded (/lib/systemd/system/kubelet.service; enabled; vendor preset: enabled)  
Drop-In: /etc/systemd/system/kubelet.service.d  
└─10-kubeadm.conf  
Active: active (running) since Wed 2017-08-23 11:01:08 HKT; 1 day 5h ago  
Docs: http://kubernetes.io/docs/  
Main PID: 3946 (kubelet)  
Tasks: 19  
Memory: 46.4M  
CPU: 20min 29.149s  
CGroup: /system.slice/kubelet.service  
└─3946 /usr/bin/kubelet --kubeconfig=/etc/kubernetes/kubelet.conf --require-kubeconfig=true  
└─3974 journalctl -k -f
```

图4-5

4.4 用例子把它们串起来

为了帮助大家更好地理解Kubernetes架构，我们部署一个应用来演示各个组件之间是如何协作的。

执行下列命令，结果如图4-6所示。

```
kubectl run httpd-app --image=httpd --replicas=2
```

```
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ kubectl run httpd-app --image=httpd --replicas=2  
deployment "httpd-app" created  
ubuntu@k8s-master:~$
```

图4-6

等待一段时间，应用部署完成，如图4-7所示。

```

ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get deployment
NAME          DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
httpd-app     2         2         2            2           2m
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get pod -o wide
NAME                                READY   STATUS    RESTARTS   AGE       IP            NODE
httpd-app-3211369089-9bgrz          1/1     Running   0          2m        10.244.1.58   k8s-node1
httpd-app-3211369089-gn6z5          1/1     Running   0          2m        10.244.2.39   k8s-node2
ubuntu@k8s-master:~$

```

图4-7

Kubernetes部署了deployment httpd-app，有两个副本Pod，分别运行在k8s-node1和k8s-node2。

详细讨论整个部署过程，如图4-8所示。

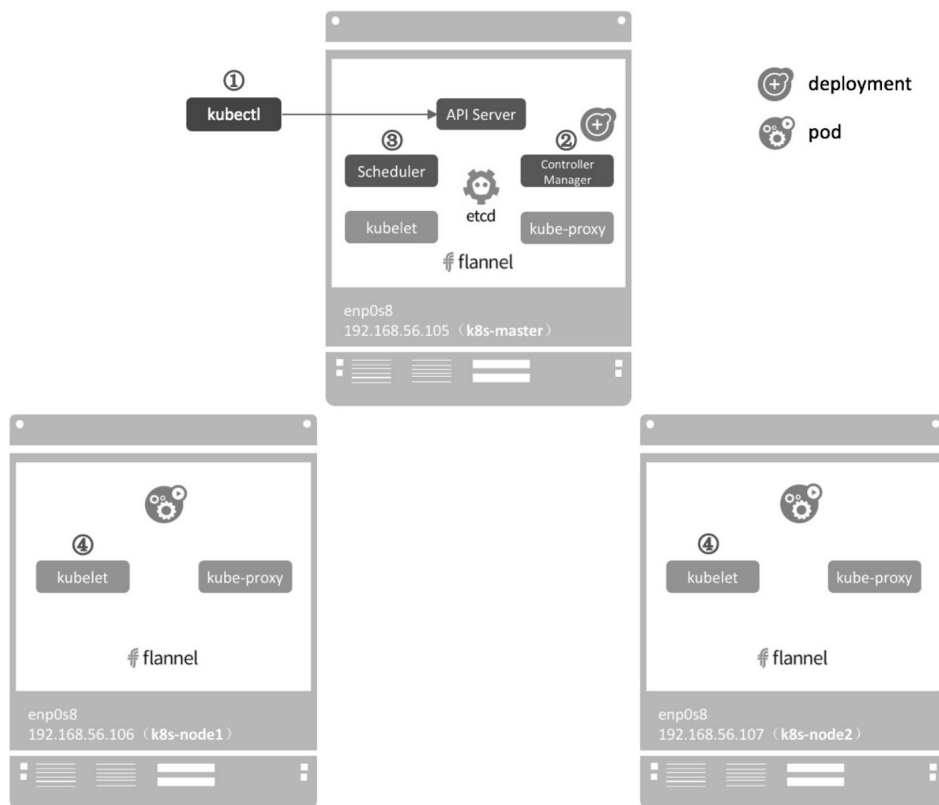


图4-8

① kubectl发送部署请求到API Server。

② API Server通知Controller Manager创建一个deployment资源。

③ Scheduler执行调度任务，将两个副本Pod分发到k8s-node1和k8s-node2。

④ k8s-node1和k8s-node2上的kubectl在各自的节点上创建并运行Pod。

补充两点：

（1）应用的配置和当前状态信息保存在etcd中，执行kubectl get pod时API Server会从etcd中读取这些数据。

（2）flannel会为每个Pod都分配IP。因为没有创建service，所以目前kube-proxy还没参与进来。

4.5 小结

本章我们学习了Kubernetes的架构，讨论了Master和Node是哪个运行的组件和服务，并通过一个部署案例加深了对架构的理解。

第5章 运行应用

从本章开始，我们将通过实践深入学习Kubernetes的各种特性。作为容器编排引擎，最重要也是最基本的功能当然是运行容器化应用，这就是本章的内容。

5.1 Deployment

前面我们已经了解到，Kubernetes通过各种Controller来管理Pod的生命周期。为了满足不同业务场景，Kubernetes开发了Deployment、ReplicaSet、DaemonSet、StatefulSet、Job等多种Controller。我们首先学习最常用的Deployment。

5.1.1 运行Deployment

先从例子开始，运行一个Deployment：

```
kubectl run nginx-deployment --image=nginx:1.7.9 --replicas=2
```

上面的命令将部署包含两个副本的Deployment nginx-deployment，容器的image为nginx:1.7.9。

下面详细分析Kubernetes都做了些什么工作，如图5-1所示。

```
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ kubectl run nginx-deployment --image=nginx:1.7.9 --replicas=2  
deployment "nginx-deployment" created  
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ kubectl get deployment nginx-deployment  
NAME           DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE  
nginx-deployment 2         2         2            2           23s  
ubuntu@k8s-master:~$
```

图5-1

在图5-1中，通过kubectl get deployment命令查看nginx-deployment的状态，输出显示两个副本正常运行。

接下来我们用kubectl describe deployment了解更详细的信息，如图5-2和图5-3所示。

```
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ kubectl describe deployment nginx-deployment  
Name:          nginx-deployment  
Namespace:     default  
CreationTimestamp: Mon, 28 Aug 2017 10:28:32 +0800  
Labels:        run=nginx-deployment  
Annotations:   deployment.kubernetes.io/revision=1  
Selector:      run=nginx-deployment  
Replicas:      2 desired | 2 updated | 2 total | 2 available | 0 unavailable  
StrategyType:  RollingUpdate  
MinReadySeconds: 0  
RollingUpdateStrategy: 1 max unavailable, 1 max surge  
Pod Template:  
  Labels:      run=nginx-deployment  
  Containers:  
    nginx-deployment:  
      Image:      nginx:1.7.9  
      Port:       <none>  
      Environment: <none>  
      Mounts:      <none>  
      Volumes:     <none>  
Conditions:  
  Type          Status  Reason  
  ----          -  
  Available     True    MinimumReplicasAvailable
```

图5-2

Type	Reason	Message
-----	-----	-----
Normal	ScalingReplicaSet	Scaled up replica set nginx-deployment-1260880958 to 2

图5-3

大部分内容都是自解释的，我们重点看图5-3。这里告诉我们创建了一个ReplicaSet nginx-deployment-1260880958，Events是Deployment的日志，记录了ReplicaSet的启动过程。通过上面的分析，也验证了Deployment通过ReplicaSet来管理Pod的事实。接着我们将注意力切换到nginx-deployment-1260880958，执行kubectl describe replicaset，如图5-4所示。

```
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get replicaset
NAME                                DESIRED    CURRENT    READY    AGE
nginx-deployment-1260880958        2          2          2        3m
ubuntu@k8s-master:~$
```

图5-4

两个副本已经就绪，用kubectl describe replicaset查看详细信息，如图5-5和图5-6所示。

```
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl describe replicaset nginx-deployment-1260880958
Name:                                nginx-deployment-1260880958
Namespace:                           default
Selector:                             pod-template-hash=1260880958,run=nginx-deployment
Labels:                               pod-template-hash=1260880958
                                      run=nginx-deployment
Annotations:                          deployment.kubernetes.io/desired-replicas=2
                                      deployment.kubernetes.io/max-replicas=3
                                      deployment.kubernetes.io/revision=1
Controlled By: Deployment/nginx-deployment
Replicas:                             2 current / 2 desired
Pods Status:                          2 Running / 0 Waiting / 0 Succeeded / 0 Failed
Pod Template:
  Labels:                             pod-template-hash=1260880958
                                      run=nginx-deployment
  Containers:
    nginx-deployment:
      Image:                           nginx:1.7.9
      Port:                            <none>
      Environment:                     <none>
      Mounts:                          <none>
```

图5-5

Type	Reason	Message
-----	-----	-----
Normal	SuccessfulCreate	Created pod: nginx-deployment-1260880958-kn8w3
Normal	SuccessfulCreate	Created pod: nginx-deployment-1260880958-rpjdc

图5-6

Controlled By指明此ReplicaSet是由Deployment nginx-deployment创建的。图5-6是两个副本Pod创建的日志。接着我们来看Pod，执行kubectl get pod，如图5-7所示。

```
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get pod
NAME                                READY    STATUS    RESTARTS   AGE
nginx-deployment-1260880958-kn8w3  1/1     Running   0           7m
nginx-deployment-1260880958-rpjdc  1/1     Running   0           7m
ubuntu@k8s-master:~$
```

图5-7

两个副本Pod都处于Running状态，然后用kubectl describe pod查看更详细的信息，如图5-8和图5-9所示。

```
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl describe pod nginx-deployment-1260880958-kn8w3
Name:          nginx-deployment-1260880958-kn8w3
Namespace:    default
Node:         k8s-node1/192.168.56.106
Start Time:   Mon, 28 Aug 2017 10:28:32 +0800
Labels:       pod-template-hash=1260880958
              run=nginx-deployment
Annotations:  kubernetes.io/created-by={"kind":"SerializedReference","apiVersion":"v1","reference":
e7-beb2-0800...
Status:       Running
IP:          10.244.1.69
Created By:   ReplicaSet/nginx-deployment-1260880958
Controlled By: ReplicaSet/nginx-deployment-1260880958
Containers:
  nginx-deployment:
    Container ID:  docker://2653d5b3aa75c16632ac2e3fc0fd5411b19d9535d47c0c50f35913334b7e0d0b
    Image:         nginx:1.7.9
    Image ID:      docker-pullable://nginx@sha256:e3456c851a152494c3e4ff5fcc26f240206abac0c9d794
    Port:         <none>
    State:         Running
      Started:     Mon, 28 Aug 2017 10:28:33 +0800
    Ready:         True
    Restart Count: 0
    Environment:   <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-slv6h (ro)
Conditions:
  Type             Status
  Initialized       True
  Ready             True
  PodScheduled      True
Volumes:
  default-token-slv6h:
    Type:          Secret (a volume populated by a Secret)
    SecretName:    default-token-slv6h
    Optional:      false
QoS Class:        BestEffort
Node-Selectors:   <none>
Tolerations:      node.alpha.kubernetes.io/notReady:NoExecute for 300s
```

图5-8

Type	Reason	Message
Normal	Scheduled	Successfully assigned nginx-deployment-1260880958-kn8w3 to k
Normal	SuccessfulMountVolume	MountVolume.SetUp succeeded for volume "default-token-s1v6h"
Normal	Pulled	Container image "nginx:1.7.9" already present on machine
Normal	Created	Created container
Normal	Started	Started container

图5-9

Controlled By指明此Pod是由ReplicaSet nginx-deployment-1260880958创建的。Events记录了Pod的启动过程。如果操作失败（比如image不存在），也能在这里查到原因。

总结一下这个过程中，如图5-10所示。

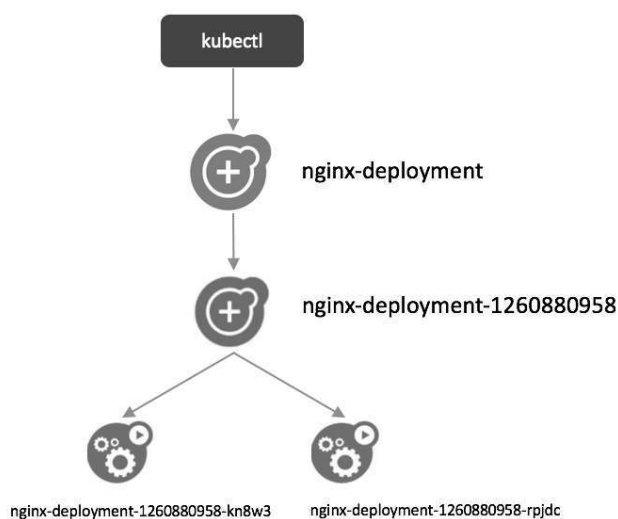


图5-10

- (1) 用户通过kubectl创建Deployment。
- (2) Deployment创建ReplicaSet。
- (3) ReplicaSet创建Pod。

从图5-10也可以看出，对象的命名方式是“子对象的名字=父对象名字+随机字符串或数字”。

5.1.2 命令vs配置文件

Kubernetes支持两种创建资源的方式：

（1）用kubect命令直接创建，比如“`kubect run nginx-deployment --image=nginx:1.7.9--replicas=2`”，在命令行中通过参数指定资源的属性。

（2）通过配置文件和kubect apply创建。要完成前面同样的工作，可执行命令“`kubect apply -f nginx.yml`”，nginx.yml的内容如图5-11所示。

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 2
  template:
    metadata:
      labels:
        app: web_server
    spec:
      containers:
        - name: nginx
          image: nginx:1.7.9
```

图5-11

资源的属性写在配置文件中，文件格式为YAML。

下面对这两种方式进行比较。

（1）基于命令的方式：

- 简单、直观、快捷，上手快。
- 适合临时测试或实验。

（2）基于配置文件的方式：

- 配置文件描述了What，即应用最终要达到的状态。
- 配置文件提供了创建资源的模板，能够重复部署。
- 可以像管理代码一样管理部署。

- 适合正式的、跨环境的、规模化部署。
- 这种方式要求熟悉配置文件的语法，有一定难度。

后面我们都将采用配置文件的方式，大家需要尽快熟悉和掌握。

`kubectl apply`不但能够创建Kubernetes资源，也能对资源进行更新，非常方便。不过Kubernetes还提供了几个类似的命令，例如`kubectl create`、`kubectl replace`、`kubectl edit`和`kubectl patch`。

为避免造成不必要的困扰，我们会尽量只使用`kubectl apply`，此命令已经能够应对百分之九十多的场景，事半功倍。

5.1.3 Deployment配置文件简介

既然要用YAML配置文件部署应用，现在就很有必要了解一下Deployment的配置格式了，其他Controller（比如DaemonSet）非常类似。

以nginx-deployment为例，配置文件如图5-12所示。

```
apiVersion: extensions/v1beta1 ①
kind: Deployment ②
metadata: ③
  name: nginx-deployment
spec: ④
  replicas: 2 ⑤
  template: ⑥
    metadata:
      labels: ⑦
        app: web_server
    spec: ⑧
      containers:
        - name: nginx
          image: nginx:1.7.9
```

图5-12

① `apiVersion`是当前配置格式的版本。

② `kind`是要创建的资源类型，这里是Deployment。

③ `metadata`是该资源的元数据，`name`是必需的元数据项。

- ④ spec部分是该Deployment的规格说明。
- ⑤ replicas指明副本数量，默认为1。
- ⑥ template定义Pod的模板，这是配置文件的重要部分。
- ⑦ metadata定义Pod的元数据，至少要定义一个label。label的key和value可以任意指定。
- ⑧ spec描述Pod的规格，此部分定义Pod中每一个容器的属性，name和image是必需的。

此nginx.yml是一个最简单的Deployment配置文件，后面我们学习Kubernetes各项功能时会逐步丰富这个文件。

执行kubect apply -f nginx.yml，如图5-13所示。

```
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ kubectl apply -f nginx.yml  
deployment "nginx-deployment" created  
ubuntu@k8s-master:~$
```

图5-13

部署成功。同样，也可以通过kubectl get查看nginx-deployment的各种资源，如图5-14所示。

```
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ kubectl get deployment  
NAME          DESIRED  CURRENT  UP-TO-DATE  AVAILABLE  AGE  
nginx-deployment  2        2        2           2          17s  
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ kubectl get replicaset  
NAME          DESIRED  CURRENT  READY  AGE  
nginx-deployment-2721169382  2        2        2      29s  
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ kubectl get pod -o wide  
NAME          READY  STATUS   RESTARTS  AGE  IP            NODE  
nginx-deployment-2721169382-b13cn  1/1    Running  0         41s  10.244.2.90   k8s-node2  
nginx-deployment-2721169382-x6f3z  1/1    Running  0         41s  10.244.1.80   k8s-node1  
ubuntu@k8s-master:~$
```

图5-14

Deployment、ReplicaSet、Pod都已经就绪。如果要删除这些资源，执行 kubectl delete deployment nginx-deployment 或者 kubectl delete -f nginx.yml，如图5-15所示。

```
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ kubectl delete -f nginx.yml  
deployment "nginx-deployment" deleted  
ubuntu@k8s-master:~$
```

图5-15

5.1.4 伸缩

伸缩是指在线增加或减少Pod的副本数。

Deployment nginx-deployment初始是两个副本，如图5-16所示。

```
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ kubectl apply -f nginx.yml  
deployment "nginx-deployment" created  
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ kubectl get pod -o wide  
NAME                                READY   STATUS    RESTARTS   AGE   IP            NODE  
nginx-deployment-2721169382-1gzzf   1/1     Running   0          8s    10.244.2.91   k8s-node2  
nginx-deployment-2721169382-pt53w   1/1     Running   0          8s    10.244.1.81   k8s-node1  
ubuntu@k8s-master:~$
```

图5-16

k8s-node1和k8s-node2上各跑了一个副本。现在修改nginx.yml文件，将副本改成5个，如图5-17所示。

```
apiVersion: extensions/v1beta1  
kind: Deployment  
metadata:  
  name: nginx-deployment  
spec:  
  replicas: 5  
  template:  
    metadata:  
      labels:  
        app: web_server  
    spec:  
      containers:  
        - name: nginx  
          image: nginx:1.7.9
```

图5-17

再次执行kubectl apply，如图5-18所示。

```
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl apply -f nginx.yml
deployment "nginx-deployment" configured
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get pod -o wide
NAME                                READY    STATUS    RESTARTS   AGE    IP             NODE
nginx-deployment-2721169382-gbjc2  1/1     Running   0          8s     10.244.2.93    k8s-node2
nginx-deployment-2721169382-lgzzf  1/1     Running   0          4m     10.244.2.91    k8s-node2
nginx-deployment-2721169382-mbrln  1/1     Running   0          8s     10.244.2.92    k8s-node2
nginx-deployment-2721169382-pt53w  1/1     Running   0          4m     10.244.1.81    k8s-node1
nginx-deployment-2721169382-s6hlx  1/1     Running   0          8s     10.244.1.82    k8s-node1
ubuntu@k8s-master:~$
```

图5-18

三个新副本被创建并调度到k8s-node1和k8s-node2上。

出于安全考虑，默认配置下Kubernetes不会将Pod调度到Master节点。如果希望将k8s-master也当作Node使用，可以执行如下命令：

```
kubectl taint node k8s-master node-role.kubernetes.io/master-
```

如果要恢复Master Only状态，执行如下命令：

```
kubectl taint node k8s-master node-role.kubernetes.io/master="" :NoSchedule
```

接下来修改配置文件，将副本数减少为3个，重新执行kubectl apply，如图5-19所示。

```
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl apply -f nginx.yml
deployment "nginx-deployment" configured
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get pod -o wide
NAME                                READY    STATUS    RESTARTS   AGE    IP             NODE
nginx-deployment-2721169382-gbjc2  1/1     Running   0          14m    10.244.2.93    k8s-node2
nginx-deployment-2721169382-lgzzf  1/1     Running   0          18m    10.244.2.91    k8s-node2
nginx-deployment-2721169382-mbrln  0/1     Terminating 0          14m    <none>         k8s-node2
nginx-deployment-2721169382-pt53w  1/1     Running   0          18m    10.244.1.81    k8s-node1
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get pod -o wide
NAME                                READY    STATUS    RESTARTS   AGE    IP             NODE
nginx-deployment-2721169382-gbjc2  1/1     Running   0          14m    10.244.2.93    k8s-node2
nginx-deployment-2721169382-lgzzf  1/1     Running   0          18m    10.244.2.91    k8s-node2
nginx-deployment-2721169382-pt53w  1/1     Running   0          18m    10.244.1.81    k8s-node1
ubuntu@k8s-master:~$
```

图5-19

可以看到两个副本被删除，最终保留了3个副本。

5.1.5 Failover

下面我们模拟k8s-node2故障，关闭该节点，如图5-20所示。

```
root@k8s-node2:~#
root@k8s-node2:~# halt -h
Connection to 192.168.56.107 closed by remote host.
Connection to 192.168.56.107 closed.
```

图5-20

等待一段时间，Kubernetes会检查到k8s-node2不可用，将k8s-node2上的Pod标记为Unknown状态，并在k8s-node1上新创建两个Pod，维持总副本数为3，如图5-21所示。

```
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get node
NAME          STATUS    AGE      VERSION
k8s-master    Ready     6d        v1.7.4
k8s-node1     Ready     5d        v1.7.4
k8s-node2     NotReady  5d        v1.7.4
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get pod -o wide
NAME          READY   STATUS    RESTARTS   AGE    IP            NODE
nginx-deployment-2721169382-gbjc2  1/1     Unknown   0           22m    10.244.2.93   k8s-node2
nginx-deployment-2721169382-lgzzf  1/1     Unknown   0           27m    10.244.2.91   k8s-node2
nginx-deployment-2721169382-p4mhr  1/1     Running   0           17s    10.244.1.84   k8s-node1
nginx-deployment-2721169382-pt53w  1/1     Running   0           27m    10.244.1.81   k8s-node1
nginx-deployment-2721169382-x8rtb  1/1     Running   0           17s    10.244.1.83   k8s-node1
ubuntu@k8s-master:~$
```

图5-21

当k8s-node2恢复后，Unknown的Pod会被删除，不过已经运行的Pod不会重新调度回k8s-node2，如图5-22所示。

```
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get node
NAME          STATUS    AGE      VERSION
k8s-master    Ready     6d        v1.7.4
k8s-node1     Ready     5d        v1.7.4
k8s-node2     Ready     5d        v1.7.4
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get pod -o wide
NAME          READY   STATUS    RESTARTS   AGE    IP            NODE
nginx-deployment-2721169382-p4mhr  1/1     Running   1           12m    10.244.1.87   k8s-node1
nginx-deployment-2721169382-pt53w  1/1     Running   1           39m    10.244.1.85   k8s-node1
nginx-deployment-2721169382-x8rtb  1/1     Running   1           12m    10.244.1.88   k8s-node1
ubuntu@k8s-master:~$
```

图5-22

删除nginx-deployment，如图5-23所示。

```
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl delete deployment nginx-deployment
deployment "nginx-deployment" deleted
ubuntu@k8s-master:~$
```

图5-23

5.1.6 用label控制Pod的位置

默认配置下，Scheduler会将Pod调度到所有可用的Node。不过有些情况我们希望将Pod部署到指定的Node，比如将有大量磁盘I/O的Pod部署到配置了SSD的Node；或者Pod需要GPU，需要运行在配置了GPU的节点上。

Kubernetes是通过label来实现这个功能的。

label是key-value对，各种资源都可以设置label，灵活添加各种自定义属性。比如执行如下命令标注k8s-node1是配置了SSD的节点。

```
kubectl label node k8s-node1 disktype=ssd
```

然后通过kubectl get node --show-labels查看节点的label，如图5-24所示。

```
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ kubectl label node k8s-node1 disktype=ssd  
node "k8s-node1" labeled  
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ kubectl get node --show-labels  
NAME          STATUS    AGE      VERSION   LABELS  
k8s-master    Ready     7d       v1.7.4    beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,kubernetes.io/host-arch=amd64,kubernetes.io/os=linux  
k8s-node1     Ready     7d       v1.7.4    beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,disktype=ssd,kubernetes.io/host-arch=amd64,kubernetes.io/os=linux  
k8s-node2     Ready     7d       v1.7.4    beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,kubernetes.io/host-arch=amd64,kubernetes.io/os=linux  
ubuntu@k8s-master:~$
```

图5-24

disktype=ssd已经成功添加到k8s-node1，除了disktype，Node还有几个Kubernetes自己维护的label。

有了disktype这个自定义label，接下来就可以指定将Pod部署到k8s-node1。编辑nginx.yml，如图5-25所示。

```

apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 6
  template:
    metadata:
      labels:
        app: web_server
    spec:
      containers:
        - name: nginx
          image: nginx:1.7.9
      nodeSelector:
        disktype: ssd

```

图5-25

在Pod模板的spec里通过nodeSelector指定将此Pod部署到具有label disktype=ssd的Node上。

部署Deployment并查看Pod的运行节点，如图5-26所示。

```

ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl apply -f nginx.yml
deployment "nginx-deployment" created
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get pod -o wide

```

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
nginx-deployment-204403116-1kk23	1/1	Running	0	23s	10.244.1.97	k8s-node1
nginx-deployment-204403116-gcnkz	1/1	Running	0	23s	10.244.1.101	k8s-node1
nginx-deployment-204403116-kmbwr	1/1	Running	0	23s	10.244.1.96	k8s-node1
nginx-deployment-204403116-kzpnf	1/1	Running	0	23s	10.244.1.100	k8s-node1
nginx-deployment-204403116-vbz47	1/1	Running	0	23s	10.244.1.98	k8s-node1
nginx-deployment-204403116-vvh54	1/1	Running	0	23s	10.244.1.99	k8s-node1

```

ubuntu@k8s-master:~$

```

图5-26

全部6个副本都运行在k8s-node1上，符合我们的预期。要删除label disktype，执行如下命令：

kubectl label node k8s-node1 disktype-

- 即删除，如图5-27所示。

```

ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl label node k8s-node1 disktype-
node "k8s-node1" labeled
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get node --show-labels

```

NAME	STATUS	AGE	VERSION	LABELS
k8s-master	Ready	8d	v1.7.4	beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,kubernetes.io/hostname=k8s-master,node-role.kubernetes.io/master=
k8s-node1	Ready	7d	v1.7.4	beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,kubernetes.io/hostname=k8s-node1
k8s-node2	Ready	7d	v1.7.4	beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,kubernetes.io/hostname=k8s-node2

```

ubuntu@k8s-master:~$

```

图5-27

不过此时Pod并不会重新部署，依然在k8s-node1上运行，如图5-28所示。

```
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get pod -o wide
NAME                                READY    STATUS    RESTARTS   AGE    IP            NODE
nginx-deployment-204403116-1kk23    1/1      Running   0           6m     10.244.1.97   k8s-node1
nginx-deployment-204403116-gcnkz    1/1      Running   0           6m     10.244.1.101  k8s-node1
nginx-deployment-204403116-kmbwr    1/1      Running   0           6m     10.244.1.96   k8s-node1
nginx-deployment-204403116-kzpnf    1/1      Running   0           6m     10.244.1.100  k8s-node1
nginx-deployment-204403116-vbz47    1/1      Running   0           6m     10.244.1.98   k8s-node1
nginx-deployment-204403116-vvh54    1/1      Running   0           6m     10.244.1.99   k8s-node1
ubuntu@k8s-master:~$
```

图5-28

除非在nginx.yml中删除nodeSelector设置，然后通过kubectl apply重新部署，如图5-29所示。

```
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl apply -f nginx.yml
deployment "nginx-deployment" configured
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get pod -o wide
NAME                                READY    STATUS    RESTARTS   AGE    IP            NODE
nginx-deployment-204403116-1kk23    0/1      Terminating 0       7m     <none>        k8s-node1
nginx-deployment-204403116-kmbwr    1/1      Terminating 0       7m     10.244.1.96   k8s-node1
nginx-deployment-204403116-kzpnf    0/1      Terminating 0       7m     <none>        k8s-node1
nginx-deployment-204403116-vbz47    0/1      Terminating 0       7m     <none>        k8s-node1
nginx-deployment-204403116-vvh54    0/1      Terminating 0       7m     <none>        k8s-node1
nginx-deployment-2721169382-56mhh    1/1      Running      0       4s     10.244.2.101  k8s-node2
nginx-deployment-2721169382-bf0rd    1/1      Running      0       6s     10.244.2.99   k8s-node2
nginx-deployment-2721169382-cnh8b    1/1      Running      0       4s     10.244.2.100  k8s-node2
nginx-deployment-2721169382-lqhn6    1/1      Running      0       6s     10.244.2.98   k8s-node2
nginx-deployment-2721169382-q61jr    1/1      Running      0       4s     10.244.1.102  k8s-node1
nginx-deployment-2721169382-r983l    1/1      Running      0       6s     10.244.2.97   k8s-node2
ubuntu@k8s-master:~$
```

图5-29

Kubernetes会删除之前的Pod并调度和运行新的Pod。

5.2 DaemonSet

Deployment部署的副本Pod会分布在各个Node上，每个Node都可能运行好几个副本。DaemonSet的不同之处在于：每个Node上最多只能运行一个副本。

DamonSet的典型应用场景有：

- (1) 在集群的每个节点上运行存储Daemon，比如glusterd或ceph。

(2) 在每个节点上运行日志收集Daemon，比如fluentd或logstash。

(3) 在每个节点上运行监控Daemon，比如Prometheus Node Exporter或collectd。

其实Kubernetes自己就在用DaemonSet运行系统组件。执行如下命令，如图5-30所示。

```
kubectl get daemonset --namespace=kube-system
```

```
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ kubectl get daemonset --namespace=kube-system  
NAME          DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE-SELECTOR  
kube-flannel-ds 3          3         3       3            3           beta.kubernetes.io/arch=amd64  
kube-proxy     3          3         3       3            3           <none>
```

图5-30

DaemonSet kube-flannel-ds和kube-proxy分别负责在每个节点上运行flannel和kube-proxy组件，如图5-31所示。

```
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ kubectl get pod --namespace=kube-system -o wide  
NAME                                READY   STATUS    RESTARTS   AGE   IP             NODE  
etcd-k8s-master                     1/1    Running   1           8d    192.168.56.105 k8s-master  
kube-apiserver-k8s-master           1/1    Running   1           8d    192.168.56.105 k8s-master  
kube-controller-manager-k8s-master 1/1    Running   1           8d    192.168.56.105 k8s-master  
kube-dns-2425271678-5lxc4          3/3    Running   6           3d    10.244.1.89    k8s-node1  
kube-flannel-ds-cqpbp               2/2    Running   11          8d    192.168.56.106 k8s-node1  
kube-flannel-ds-v0p3x               2/2    Running   3           8d    192.168.56.105 k8s-master  
kube-flannel-ds-xk49w               2/2    Running   12          8d    192.168.56.107 k8s-node2  
kube-proxy-16mg9                    1/1    Running   3           8d    192.168.56.106 k8s-node1  
kube-proxy-wc4j0                    1/1    Running   1           8d    192.168.56.105 k8s-master  
kube-proxy-xl5gd                    1/1    Running   4           8d    192.168.56.107 k8s-node2  
kube-scheduler-k8s-master           1/1    Running   1           8d    192.168.56.105 k8s-master  
ubuntu@k8s-master:~$
```

图5-31

因为flannel和kube-proxy属于系统组件，需要在命令行中通过--namespace=kube-system指定namespace kube-system。若不指定，则只返回默认namespace default中的资源。

5.2.1 kube-flannel-ds

下面我们通过分析kube-flannel-ds来学习DaemonSet。

还记得之前是如何部署flannel网络的吗？我们执行了如下命令：

```
kubectl apply -f https://raw.githubusercontent.com/coreos/flannel/master/Documentation/kube-flannel.yml
```

flannel的DaemonSet就定义在kube-flannel.yml中，如图5-32所示。

```
apiVersion: extensions/v1beta1
kind: DaemonSet ①
metadata:
  name: kube-flannel-ds
  namespace: kube-system
  labels:
    tier: node
    app: flannel
spec:
  template:
    metadata:
      labels:
        tier: node
        app: flannel
    spec:
      hostNetwork: true ②
      nodeSelector:
        beta.kubernetes.io/arch: amd64
      containers: ③
      - name: kube-flannel
        image: quay.io/coreos/flannel:v0.8.0-amd64
        command: [ "/opt/bin/flanneld", "--ip-masq", "--kube-subnet-mgr" ]
      - name: install-cni
        image: quay.io/coreos/flannel:v0.8.0-amd64
        command: [ "/bin/sh", "-c", "set -e -x; cp -f /etc/kube-flannel/cni-conf.json
```

图5-32

注意：配置文件的完整内容要更复杂一些，为了更好地学习DaemonSet，这里只保留了最重要的内容。

① DaemonSet配置文件的语法和结构与Deployment几乎完全一样，只是将kind设为DaemonSet。

② hostName指定Pod直接使用Node网络，相当于docker run --network=host。考虑到flannel需要为集群提供网络连接，这个要求是合理的。

③ containers定义了运行flannel服务的两个容器。

下面我们再来分析另一个DaemonSet：kube-proxy。

5.2.2 kube-proxy

由于无法拿到kube-proxy的YAML文件，只能运行如下命令查看配置：

```
kubectl edit daemonset kube-proxy --namespace=kube-system
```

结果如图5-33所示。

```
apiVersion: extensions/v1beta1
kind: DaemonSet ①
metadata:
  labels:
    k8s-app: kube-proxy
  name: kube-proxy
  namespace: kube-system
spec:
  selector:
    matchLabels:
      k8s-app: kube-proxy
  template:
    metadata:
      labels:
        k8s-app: kube-proxy
    spec:
      containers: ②
      - command:
        - /usr/local/bin/kube-proxy
        - --kubeconfig=/var/lib/kube-proxy/kubeconfig.conf
        - --cluster-cidr=10.244.0.0/16
        image: gcr.io/google_containers/kube-proxy-amd64:v1.7.4
        name: kube-proxy
status: ③
  currentNumberScheduled: 3
  desiredNumberScheduled: 3
  numberAvailable: 3
  numberMisscheduled: 0
  numberReady: 3
  observedGeneration: 1
  updatedNumberScheduled: 3
```

图5-33

同样为了便于理解，这里只保留了最重要的信息。

① kind: DaemonSet指定这是一个DaemonSet类型的资源。

② containers定义了kube-proxy的容器。

③ status是当前DaemonSet的运行时状态，这个部分是kubectl edit特有的。其实Kubernetes集群中每个当前运行的资源都可以通过kubectl edit

查看其配置和运行状态，比如 `kubectl edit deployment nginx-deployment`。

5.2.3 运行自己的DaemonSet

本小节以 Prometheus Node Exporter 为例演示用户如何运行自己的 DaemonSet。

Prometheus 是流行的系统监控方案，Node Exporter 是 Prometheus 的 agent，以 Daemon 的形式运行在每个被监控节点上。

如果是直接在 Docker 中运行 Node Exporter 容器，命令为：

```
docker run -d \

-v "/proc:/host/proc" \

-v "/sys:/host/sys" \

-v "":"/rootfs" \

--net=host \

prom/node-exporter \

--path.procfs /host/proc \

--path.sysfs /host/sys \

--collector.filesystem.ignored-mount-
points "^/(sys|proc|dev|host|etc)($|/)"
```

将其转换为 DaemonSet 的 YAML 配置文件 `node_exporter.yml`，如图 5-34 所示。

```

apiVersion: extensions/v1beta1
kind: DaemonSet
metadata:
  name: node-exporter-daemonset
spec:
  template:
    metadata:
      labels:
        app: prometheus
    spec:
      hostNetwork: true ①
      containers:
        - name: node-exporter
          image: prom/node-exporter
          imagePullPolicy: IfNotPresent
          command: ②
            - /bin/node_exporter
            - --path.procfs
            - /host/proc
            - --path.sysfs
            - /host/sys
            - --collector.filesystem.ignored-mount-points
            - ^/(sys|proc|dev|host|etc)($|/)
          volumeMounts: ③
            - name: proc
              mountPath: /host/proc
            - name: sys
              mountPath: /host/sys
            - name: root
              mountPath: /rootfs
          volumes:
            - name: proc
              hostPath:
                path: /proc
            - name: sys
              hostPath:
                path: /sys
            - name: root
              hostPath:
                path: /

```

图5-34

① 直接使用Host的网络。

② 设置容器启动命令。

③ 通过Volume将Host路径/proc、/sys和/映射到容器中。我们将在后面详细讨论Volume。

执行 `kubectl apply -f node_exporter.yml`，如图5-35所示。

```

ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl apply -f node_exporter.yml
daemonset "node-exporter-daemonset" created
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get pod -o wide
NAME                                READY   STATUS    RESTARTS   AGE   IP            NODE
node-exporter-daemonset-b2w0x      1/1     Running   0           6s    192.168.56.107 k8s-node2
node-exporter-daemonset-kvmkr      1/1     Running   0           6s    192.168.56.106 k8s-node1
ubuntu@k8s-master:~$

```

图5-35

DaemonSet node-exporter-daemonset部署成功，k8s-node1和k8s-node2上分别运行了一个node exporter Pod。

5.3 Job

容器按照持续运行的时间可分为两类：服务类容器和工作类容器。

服务类容器通常持续提供服务，需要一直运行，比如HTTP Server、Daemon等。工作类容器则是一次性任务，比如批处理程序，完成后容器就退出。

Kubernetes的Deployment、ReplicaSet和DaemonSet都用于管理服务类容器；对于工作类容器，我们使用Job。

先看一个简单的Job配置文件myjob.yml，如图5-36所示。

```

apiVersion: batch/v1 ①
kind: Job ②
metadata:
  name: myjob
spec:
  template:
    metadata:
      name: myjob
    spec:
      containers:
      - name: hello
        image: busybox
        command: ["echo", "hello k8s job! "]
        restartPolicy: Never ③

```

图5-36

① batch/v1是当前Job的apiVersion。

② 指明当前资源的类型为Job。

③ restartPolicy指定什么情况下需要重启容器。对于Job，只能设置为Never或者OnFailure。对于其他controller（比如Deployment），可以设置为Always。

通过kubect apply -f myjob.yml启动Job，如图5-37所示。

```
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ kubectl apply -f myjob.yml  
job "myjob" created  
ubuntu@k8s-master:~$
```

图5-37

通过kubectl get job查看Job的状态，如图5-38所示。

```
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ kubectl get job  
NAME          DESIRED  SUCCESSFUL  AGE  
myjob         1        1           12s  
ubuntu@k8s-master:~$
```

图5-38

DESIRED和SUCCESSFUL都为1，表示按照预期启动了一个Pod，并且已经成功执行。通过kubectl get pod查看Pod的状态，如图5-39所示。

```
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ kubectl get pod  
No resources found, use --show-all to see completed objects.  
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ kubectl get pod --show-all  
NAME          READY    STATUS    RESTARTS  AGE  
myjob-nfkxk   0/1     Completed  0         2m  
ubuntu@k8s-master:~$
```

图5-39

因为Pod执行完毕后容器已经退出，需要用--show-all才能查看Completed状态的Pod。

通过kubectl logs可以查看Pod的标准输出，如图5-40所示。

```
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ kubectl logs myjob-nfkxx  
hello k8s job!  
ubuntu@k8s-master:~$
```

图5-40

5.3.1 Pod失败的情况

以上是Pod成功执行的情况，如果Pod失败了会怎么样呢？

我们做个试验，修改myjob.yml，故意引入一个错误，如图5-41所示。

```
apiVersion: batch/v1  
kind: Job  
metadata:  
  name: myjob  
spec:  
  template:  
    metadata:  
      name: myjob  
    spec:  
      containers:  
      - name: hello  
        image: busybox  
        command: ["invalid_command", "hello k8s job!"]  
      restartPolicy: Never
```

图5-41

先删除之前的Job，如图5-42所示。

```
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ kubectl delete -f myjob.yml  
job "myjob" deleted  
ubuntu@k8s-master:~$
```

图5-42

运行新的Job并查看状态，如图5-43所示。

```
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ kubectl apply -f myjob.yml  
job "myjob" created  
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ kubectl get job  
NAME          DESIRED  SUCCESSFUL  AGE  
myjob         1         0           7s  
ubuntu@k8s-master:~$
```

图5-43

当前SUCCESSFUL的Pod数量为0，查看Pod的状态，如图5-44所示。

```
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get pod --show-all
```

NAME	READY	STATUS	RESTARTS	AGE
myjob-0x0z0	0/1	ContainerCannotRun	0	11s
myjob-7f693	0/1	ContainerCannotRun	0	9s
myjob-lv7w3	0/1	ContainerCannotRun	0	6s
myjob-nm1lz	0/1	ContainerCannotRun	0	14s
myjob-qgvn0	0/1	ContainerCannotRun	0	4s
myjob-rbs1x	0/1	ContainerCreating	0	1s

```
ubuntu@k8s-master:~$
```

图5-44

可以看到有多个Pod，状态均不正常。通过kubectl describe pod查看某个Pod的启动日志，如图5-45所示。

SubObjectPath	Type	Reason	Message
-----	Normal	Scheduled	Successfully assigned myjob-0x0z0 to k8s-node2
	Normal	SuccessfulMountVolume	MountVolume.SetUp succeeded for volume "default-token-k87vh"
spec.containers(hello)	Normal	Pulling	pulling image "busybox"
spec.containers(hello)	Normal	Pulled	Successfully pulled image "busybox"
spec.containers(hello)	Normal	Created	Created container
spec.containers(hello)	Warning	Failed	Error: failed to start container "hello": executable not found in \$PATH
	Warning	FailedSync	Error syncing pod

图5-45

日志显示没有可执行程序，符合我们的预期。

下面解释一个现象：为什么kubectl get pod会看到这么多个失败的Pod？

原因是：当第一个Pod启动时，容器失败退出，根据restartPolicy: Never，此失败容器不会被重启，但Job DESIRED的Pod是1，目前SUCCESSFUL为0，不满足，所以Job controller会启动新的Pod，直到SUCCESSFUL为1。对于我们这个例子，SUCCESSFUL永远也到不了1，所以Job controller会一直创建新的Pod。为了终止这个行为，只能删除Job，如图5-46所示。

```
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl delete -f myjob.yml
job "myjob" deleted
ubuntu@k8s-master:~$
```

图5-46

如果将restartPolicy设置为OnFailure会怎么样？下面我们实践一下，修改myjob.yml后重新启动，如图5-47所示。

```
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ kubectl apply -f myjob.yml  
job "myjob" created  
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ kubectl get job  
NAME          DESIRED  SUCCESSFUL  AGE  
myjob         1        0           5s  
ubuntu@k8s-master:~$
```

图5-47

Job的SUCCESSFUL Pod数量还是0，再看看Pod的情况，如图5-48所示。

```
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ kubectl get pod --show-all  
NAME          READY   STATUS    RESTARTS   AGE  
myjob-83lx0   0/1     CrashLoopBackOff  3          1m  
ubuntu@k8s-master:~$
```

图5-48

这里只有一个Pod，不过RESTARTS为3，而且不断增加，说明OnFailure生效，容器失败后会自动重启。

5.3.2 Job的并行性

有时我们希望能同时运行多个Pod，提高Job的执行效率。这个可以通过parallelism设置，如图5-49所示。

```
apiVersion: batch/v1  
kind: Job  
metadata:  
  name: myjob  
spec:  
  parallelism: 2  
  template:  
    metadata:  
      name: myjob  
    spec:  
      containers:  
      - name: hello  
        image: busybox  
        command: ["echo", "hello k8s job!"]  
      restartPolicy: OnFailure
```

图5-49

这里我们将并行的Pod数量设置为2，实践一下，如图5-50所示。Job一共启动了两个Pod，而且AGE相同，可见是并行运行的。

```
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ kubectl apply -f myjob.yml  
job "myjob" created  
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ kubectl get job  
NAME      DESIRED  SUCCESSFUL  AGE  
myjob     <none>    2           6s  
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ kubectl get pod --show-all -o wide  
NAME      READY   STATUS    RESTARTS  AGE   IP            NODE  
myjob-cn4zs  0/1     Completed  0         12s   10.244.2.5    k8s-node2  
myjob-vhpzs  0/1     Completed  0         12s   10.244.1.28   k8s-node1  
ubuntu@k8s-master:~$
```

图5-50

我们还可以通过completions设置Job成功完成Pod的总数，如图5-51所示。

```
apiVersion: batch/v1  
kind: Job  
metadata:  
  name: myjob  
spec:  
  completions: 6  
  parallelism: 2  
  template:  
    metadata:  
      name: myjob  
    spec:  
      containers:  
      - name: hello  
        image: busybox  
        command: ["echo", "hello k8s job!"]  
      restartPolicy: OnFailure
```

图5-51

上面配置的含义是：每次运行两个Pod，直到总共有6个Pod成功完成。实践一下，如图5-52所示。

```

ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl apply -f myjob.yml
job "myjob" created
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get job
NAME          DESIRED  SUCCESSFUL  AGE
myjob         6         6            9s
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get pod --show-all -o wide
NAME          READY   STATUS    RESTARTS  AGE    IP             NODE
myjob-0t4zk   0/1     Completed  0          16s    10.244.2.6     k8s-node2
myjob-79xkx   0/1     Completed  0          11s    10.244.2.8     k8s-node2
myjob-p42lx   0/1     Completed  0          14s    10.244.1.30    k8s-node1
myjob-rfvd7   0/1     Completed  0          13s    10.244.2.7     k8s-node2
myjob-srpg9   0/1     Completed  0          12s    10.244.1.31    k8s-node1
myjob-wl8tt   0/1     Completed  0          16s    10.244.1.29    k8s-node1
ubuntu@k8s-master:~$

```

图5-52

DESIRED和SUCCESSFUL均为6，符合预期。如果不指定completions和parallelism，默认值均为1。

上面的例子只是为了演示Job的并行特性，实际用途不大。不过现实中确实存在很多需要并行处理的场景。比如批处理程序，每个副本（Pod）都会从任务池中读取任务并执行，副本越多，执行时间就越短，效率就越高。这种类似的场景都可以用Job来实现。

5.3.3 定时Job

Linux中有cron程序定时执行任务，Kubernetes的CronJob提供了类似的功能，可以定时执行Job。CronJob配置文件示例如图5-53所示。

```

apiVersion: batch/v2alpha1 ①
kind: CronJob ②
metadata:
  name: hello
spec:
  schedule: "*/1 * * * *" ③
  jobTemplate: ④
    spec:
      template:
        spec:
          containers:
            - name: hello
              image: busybox
              command: ["echo", "hello k8s job! "]
              restartPolicy: OnFailure

```

图5-53

- ① batch/v2alpha1是当前CronJob的apiVersion。
- ② 指明当前资源的类型为CronJob。
- ③ schedule指定什么时候运行Job，其格式与Linux cron一致。这里*/1 * * * *的含义是每一分钟启动一次。
- ④ jobTemplate定义Job的模板，格式与前面的Job一致。

接下来通过kubectl apply创建CronJob，如图5-54所示。

```
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ kubectl apply -f cronjob.yml  
error: error validating "cronjob.yml": error validating data: the server could not find the  
resource; if you choose to ignore these errors, turn validation off with --validate=false  
ubuntu@k8s-master:~$
```

图5-54

失败了。这是因为Kubernetes默认没有enable CronJob功能，需要在kube-apiserver中加入这个功能。方法很简单，修改kube-apiserver的配置文件/etc/kubernetes/manifests/kubeapiserver.yaml，如图5-55所示。

```
apiVersion: v1  
kind: Pod  
metadata:  
  annotations:  
    scheduler.alpha.kubernetes.io/critical-pod: ""  
  creationTimestamp: null  
  labels:  
    component: kube-apiserver  
    tier: control-plane  
  name: kube-apiserver  
  namespace: kube-system  
spec:  
  containers:  
    - command:  
      - kube-apiserver  
      - --runtime-config=batch/v2alpha1=true  
      - --requestheader-group-headers=X-Remote-Group  
      - --requestheader-extra-headers-prefix=X-Remote-Extra-  
      - --requestheader-allowed-names=front-proxy-client  
      - --kubelet-client-key=/etc/kubernetes/pki/apiserver-kubelet-client  
      - --kubelet-preferred-address-types=InternalIP,ExternalIP,Hostname  
      - --allow-privileged=true  
      - --experimental-bootstrap-token-auth=true  
      - --requestheader-username-headers=X-Remote-User  
      - --service-account-key-file=/etc/kubernetes/pki/sa.pub
```

图5-55

kube-apiserver 本身也是一个 Pod，在启动参数中加上 `--runtime-config=batch/v2alpha1=true` 即可。然后重启 kubelet 服务：

```
systemctl restart kubelet.service
```

kubelet 会重启 kube-apiserver Pod。通过 `kubectl api-versions` 确认 kube-apiserver 现在已经支持 batch/v2alpha1，如图 5-56 所示。

```
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ kubectl api-versions  
apiextensions.k8s.io/v1beta1  
apiregistration.k8s.io/v1beta1  
apps/v1beta1  
authentication.k8s.io/v1  
authentication.k8s.io/v1beta1  
authorization.k8s.io/v1  
authorization.k8s.io/v1beta1  
autoscaling/v1  
batch/v1  
batch/v2alpha1  
certificates.k8s.io/v1beta1  
extensions/v1beta1  
networking.k8s.io/v1  
policy/v1beta1  
rbac.authorization.k8s.io/v1alpha1  
rbac.authorization.k8s.io/v1beta1  
settings.k8s.io/v1alpha1  
storage.k8s.io/v1  
storage.k8s.io/v1beta1  
v1  
ubuntu@k8s-master:~$
```

图5-56

再次创建 CronJob，如图 5-57 所示。

```
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ kubectl apply -f cronjob.yml  
cronjob "hello" created  
ubuntu@k8s-master:~$
```

图5-57

这次成功了。通过 `kubectl get cronjob` 查看 CronJob 的状态，如图 5-58 所示。

```
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get cronjob
NAME          SCHEDULE          SUSPEND   ACTIVE   LAST-SCHEDULE
hello         */1 * * * *       False     0        Tue, 12 Sep 2017 10:21:00 +0800
ubuntu@k8s-master:~$
```

图5-58

等待几分钟，然后通过 `kubectl get jobs` 查看 Job 的执行情况，如图5-59所示。

```
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get jobs
NAME                DESIRED   SUCCESSFUL   AGE
hello-1505181600    1         1            5m
hello-1505181660    1         1            4m
hello-1505181720    1         1            3m
hello-1505181780    1         1            2m
hello-1505181840    1         1            1m
hello-1505181900    1         1            3s
ubuntu@k8s-master:~$
```

图5-59

可以看到每隔一分钟就会启动一个 Job。执行 `kubectl logs` 可查看某个 Job 的运行日志，如图5-60所示。

```
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get pod --show-all
NAME                READY     STATUS    RESTARTS   AGE
hello-1505181600-0hrx5  0/1      Completed  0          5m
hello-1505181660-nbgwd  0/1      Completed  0          4m
hello-1505181720-lf350  0/1      Completed  0          3m
hello-1505181780-tjhg2  0/1      Completed  0          2m
hello-1505181840-lqdbv  0/1      Completed  0          1m
hello-1505181900-ggfr2  0/1      Completed  0          10s
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl logs hello-1505181720-lf350
hello k8s job!
ubuntu@k8s-master:~$
```

图5-60

5.4 小结

运行容器化应用是 Kubernetes 最重要的核心功能。为满足不同的业务需要，Kubernetes 提供了多种 Controller，包括 Deployment、DaemonSet、Job、CronJob 等。本章我们通过实践详细学习了这些 Controller，并讨论了它们的特性和应用场景。

第6章 通过Service访问Pod

我们不应该期望Kubernetes Pod是健壮的，而是要假设Pod中的容器很可能因为各种原因发生故障而死掉。Deployment等Controller会通过动态创建和销毁Pod来保证应用整体的健壮性。换句话说，Pod是脆弱的，但应用是健壮的。

每个Pod都有自己的IP地址。当Controller用新Pod替代发生故障的Pod时，新Pod会分配到新的IP地址。这样就产生了一个问题：

如果一组Pod对外提供服务（比如HTTP），它们的IP很有可能发生变化，那么客户端如何找到并访问这个服务呢？

Kubernetes给出的解决方案是Service。

6.1 创建Service

Kubernetes Service从逻辑上代表了一组Pod，具体是哪些Pod则是由label来挑选的。Service有自己的IP，而且这个IP是不变的。客户端只需要访问Service的IP，Kubernetes则负责建立和维护Service与Pod的映射关系。无论后端Pod如何变化，对客户端不会有任何影响，因为Service没有变。

来看个例子，创建下面的这个Deployment，如图6-1所示。

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: httpd
spec:
  replicas: 3
  template:
    metadata:
      labels:
        run: httpd
    spec:
      containers:
        - name: httpd
          image: httpd
          ports:
            - containerPort: 80
```

图6-1

我们启动了三个Pod，运行httpd镜像，label是run: httpd，Service将会用这个label来挑选Pod，如图6-2所示。

```
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl apply -f httpd.yml
deployment "httpd" created
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get pod -o wide
```

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
httpd-741508562-192vp	1/1	Running	0	1m	10.244.4.5	k8s-node1
httpd-741508562-6g4fc	1/1	Running	0	1m	10.244.4.4	k8s-node1
httpd-741508562-6hh9g	1/1	Running	0	1m	10.244.5.4	k8s-node2

```
ubuntu@k8s-master:~$
```

图6-2

Pod分配了各自的IP，这些IP只能被Kubernetes Cluster中的容器和节点访问，如图6-3所示。

```
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ curl 10.244.4.5
<html><body><h1>It works!</h1></body></html>
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ curl 10.244.5.4
<html><body><h1>It works!</h1></body></html>
ubuntu@k8s-master:~$
```

图6-3

接下来创建Service，其配置文件如图6-4所示。

```
apiVersion: v1 ①
kind: Service ②
metadata:
  name: httpd-svc ③
spec:
  selector:
    run: httpd ④
  ports:
    - protocol: TCP ⑤
      port: 8080
      targetPort: 80
```

图6-4

- ① v1是Service的apiVersion。
- ② 指明当前资源的类型为Service。
- ③ Service的名字为httpd-svc。
- ④ selector指明挑选那些label为run: httpd的Pod作为Service的后端。
- ⑤ 将Service的8080端口映射到Pod的80端口，使用TCP协议。

执行kubectl apply创建Service httpd-svc，如图6-5所示。

```
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl apply -f httpd-svc.yml
service "httpd-svc" created
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get service
NAME          CLUSTER-IP    EXTERNAL-IP    PORT(S)    AGE
httpd-svc     10.99.229.179 <none>         8080/TCP   7s
kubernetes    10.96.0.1     <none>         443/TCP    2d
ubuntu@k8s-master:~$
```

图6-5

httpd-svc分配到一个CLUSTER-IP 10.99.229.179。可以通过该IP访问后端的httpd Pod，如图6-6所示。

```
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ curl 10.99.229.179:8080  
<html><body><h1>It works!</h1></body></html>  
ubuntu@k8s-master:~$
```

图6-6

根据前面的端口映射，这里要使用8080端口。另外，除了我们创建的httpd-svc，还有一个Service kubernetes，Cluster内部通过这个Service访问Kubernetes API Server。

通过kubectl describe可以查看httpd-svc与Pod的对应关系，如图6-7所示。

```
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ kubectl describe service httpd-svc  
Name:          httpd-svc  
Namespace:     default  
Labels:        <none>  
Annotations:   kubectl.kubernetes.io/last-applied-configuration:  
:8080,"protocol":"TC...  
Selector:      run=httpd  
Type:          ClusterIP  
IP:            10.99.229.179  
Port:          <unset> 8080/TCP  
Endpoints:     10.244.4.4:80,10.244.4.5:80,10.244.5.4:80  
Session Affinity: None  
Events:        <none>  
ubuntu@k8s-master:~$
```

图6-7

Endpoints罗列了三个Pod的IP和端口。我们知道Pod的IP是在容器中配置的，那么Service的Cluster IP又是配置在哪里的呢？CLUSTER-IP又是如何映射到Pod IP的呢？

答案是iptables。

6.2 Cluster IP底层实现

Cluster IP是一个虚拟IP，是由Kubernetes节点上的iptables规则管理的。

可以通过iptables-save命令打印出当前节点的iptables规则，因为输出较多，这里只截取与httpd-svc Cluster IP 10.99.229.179相关的信息，如图6-8所示。

```
-A KUBE-SERVICES ! -s 10.244.0.0/16 -d 10.99.229.179/32 -p tcp -m comment --comment "default/httpd-svc: Cluster IP" -m tcp -dport 8080 -j KUBE-MARK-MASQ
-A KUBE-SERVICES -d 10.99.229.179/32 -p tcp -m comment --comment "default/httpd-svc: Cluster IP" -m tcp -dport 8080 -j KUBE-SVC-RL3JAE4GN7VOGDGP
```

图6-8

这两条规则的含义是：

（1）如果Cluster内的Pod（源地址来自10.244.0.0/16）要访问httpd-svc，则允许。

（2）其他源地址访问httpd-svc，跳转到规则KUBE-SVC-RL3JAE4GN7VOGDGP。KUBE-SVC-RL3JAE4GN7VOGDGP规则如图6-9所示。

```
-A KUBE-SVC-RL3JAE4GN7VOGDGP -m comment --comment "default/httpd-svc: " -m statistic --mode random --probability 0.33332999982 -j KUBE-SEP-C5KB52P4BBJQ35PH
```

```
-A KUBE-SVC-RL3JAE4GN7VOGDGP -m comment --comment "default/httpd-svc: " -m statistic --mode random --probability 0.33332999982 -j KUBE-SEP-C5KB52P4BBJQ35PH
-A KUBE-SVC-RL3JAE4GN7VOGDGP -m comment --comment "default/httpd-svc: " -m statistic --mode random --probability 0.50000000000 -j KUBE-SEP-HGVKQQZZCF7RV4IT
-A KUBE-SVC-RL3JAE4GN7VOGDGP -m comment --comment "default/httpd-svc: " -j KUBE-SEP-XE25WGVXLHEIRVO5
```

图6-9

（1）1/3的概率跳转到规则KUBE-SEP-C5KB52P4BBJQ35PH。

（2）1/3的概率（剩下2/3的一半）跳转到规则KUBE-SEP-HGVKQQZZCF7RV4IT。

（3）1/3的概率跳转到规则KUBE-SEP-XE25WGVXLHEIRVO5。

上面三个跳转的规则如图6-10所示。

```

-A KUBE-SEP-CSK852P48BJQ35PH -s 10.244.4.4/32 -m comment --comment "default/httpd-svc:" -j KUBE-MARK-MASQ
-A KUBE-SEP-CSK852P48BJQ35PH -p tcp -m comment --comment "default/httpd-svc:" -m tcp -j DNAT --to-destination 10.244.4.4:80
-A KUBE-SEP-HGVKQZ2ZCF7RV4IT -s 10.244.4.5/32 -m comment --comment "default/httpd-svc:" -j KUBE-MARK-MASQ
-A KUBE-SEP-HGVKQZ2ZCF7RV4IT -p tcp -m comment --comment "default/httpd-svc:" -m tcp -j DNAT --to-destination 10.244.4.5:80
-A KUBE-SEP-XE25WGVLHEIRV05 -s 10.244.5.4/32 -m comment --comment "default/httpd-svc:" -j KUBE-MARK-MASQ
-A KUBE-SEP-XE25WGVLHEIRV05 -p tcp -m comment --comment "default/httpd-svc:" -m tcp -j DNAT --to-destination 10.244.5.4:80

```

图6-10

即将请求分别转发到后端的三个Pod。通过上面的分析，我们得到结论：**iptables**将访问**Service**的流量转发到后端**Pod**，而且使用类似轮询的负载均衡策略。

另外，需要补充一点：**Cluster**的每一个节点都配置了相同的**iptables**规则，这样就确保了整个**Cluster**都能够通过**Service**的**Cluster IP**访问**Service**，如图6-11所示。

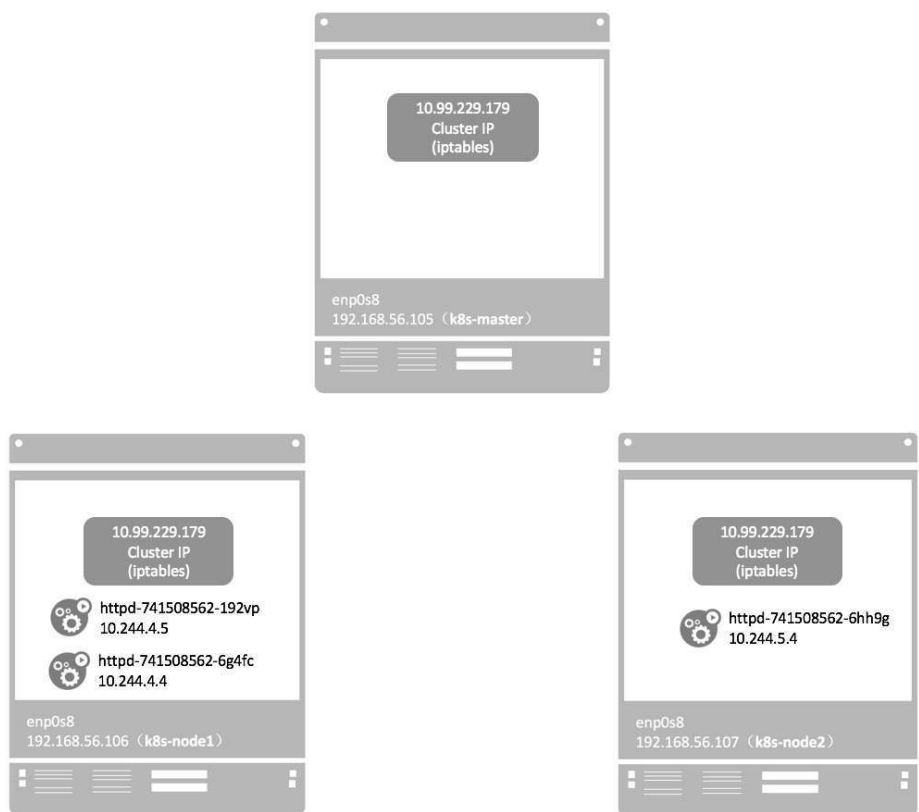


图6-11

6.3 DNS访问Service

在Cluster中，除了可以通过Cluster IP访问Service，Kubernetes还提供了更为方便的DNS访问。

kubeadm部署时会默认安装kube-dns组件，如图6-12所示。

```
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ kubectl get deployment --namespace=kube-system  
NAME          DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE  
kube-dns      1         1         1            1           5d  
ubuntu@k8s-master:~$
```

图6-12

kube-dns是一个DNS服务器。每当有新的Service被创建，kube-dns会添加该Service的DNS记录。Cluster中的Pod可以通过<SERVICE_NAME>.<NAMESPACE_NAME>访问Service。

比如可以用httpd-svc.default访问Service httpd-svc，如图6-13所示。

```
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ kubectl run busybox --rm -ti --image=busybox /bin/sh  
If you don't see a command prompt, try pressing enter.  
/ #  
/ # wget httpd-svc.default:8080  
Connecting to httpd-svc.default:8080 (10.99.229.179:8080)  
index.html 100% |*****| 45 0:00:00 ETA  
/ #
```

图6-13

如上所示，我们在一个临时的busybox Pod中验证了DNS的有效性。另外，由于这个Pod与httpd-svc同属于default namespace，因此可以省略default直接用httpd-svc访问Service，如图6-14所示。

```
/ #  
/ # wget httpd-svc:8080  
Connecting to httpd-svc:8080 (10.99.229.179:8080)  
index.html 100% |*****| 45 0:00:00 ETA  
/ #
```

图6-14

用nslookup查看httpd-svc的DNS信息，如图6-15所示。

```
/ #  
/ # nslookup httpd-svc  
Server: 10.96.0.10  
Address 1: 10.96.0.10 kube-dns.kube-system.svc.cluster.local  
  
Name: httpd-svc  
Address 1: 10.99.229.179 httpd-svc.default.svc.cluster.local  
/ #
```

图6-15

DNS 服务器是 `kube-dns.kube-system.svc.cluster.local`，这实际上就是 `kube-dns` 组件，它本身是部署在 `kube-system namespace` 中的一个 Service。

`httpd-svc.default.svc.cluster.local` 是 `httpd-svc` 的完整域名。

如果要访问其他 `namespace` 中的 Service，就必须带上 `namespace` 了。
`kubectl get namespace` 查看已有的 `namespace`，如图6-16所示。

```
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ kubectl get namespace  
NAME           STATUS    AGE  
default        Active    5d  
kube-public     Active    5d  
kube-system     Active    5d  
ubuntu@k8s-master:~$
```

图6-16

在 `kube-public` 中部署 Service `httpd2-svc`，配置如图6-17所示。

```

apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: httpd2
  namespace: kube-public
spec:
  replicas: 3
  template:
    metadata:
      labels:
        run: httpd2
    spec:
      containers:
      - name: httpd2
        image: httpd
        ports:
        - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: httpd2-svc
  namespace: kube-public
spec:
  selector:
    run: httpd2
  ports:
  - protocol: TCP
    port: 8080
    targetPort: 80

```

图6-17

通过namespace: kube-public指定资源所属的namespace。多个资源可以在一个YAML文件中定义，用“---”分割。执行kubectl apply创建资源，如图6-18所示。

```

ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl apply -f httpd2.yml
deployment "httpd2" created
service "httpd2-svc" created
ubuntu@k8s-master:~$

```

图6-18

查看kube-public的Service，如图6-19所示。

```
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ kubectl get service --namespace=kube-public  
NAME          CLUSTER-IP    EXTERNAL-IP    PORT(S)    AGE  
httpd2-svc    10.103.198.189 <none>        8080/TCP    2m  
ubuntu@k8s-master:~$
```

图6-19

在busybox Pod中访问httpd2-svc，如图6-20所示。

```
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ kubectl run busybox --rm -ti --image=busybox /bin/sh  
If you don't see a command prompt, try pressing enter.  
/#  
/# wget wget httpd2-svc:8080  
wget: bad address 'wget'  
/#  
/# wget httpd2-svc.kube-public:8080  
Connecting to httpd2-svc.kube-public:8080 (10.103.198.189:8080)  
index.html      100% |*****| 45 0:00:00 ETA  
/#
```

图6-20

因为不属于同一个namespace，所以必须使用httpd2-svc.kube-public才能访问到。

6.4 外网如何访问Service

除了Cluster内部可以访问Service，很多情况下我们也希望应用的Service能够暴露给Cluster外部。Kubernetes提供了多种类型的Service，默认是ClusterIP。

(1) ClusterIP

Service通过Cluster内部的IP对外提供服务，只有Cluster内的节点和Pod可访问，这是默认的Service类型，前面实验中的Service都是ClusterIP。

(2) NodePort

Service通过Cluster节点的静态端口对外提供服务。Cluster外部可以通过<NodeIP>:<NodePort>访问Service。

(3) LoadBalancer

Service利用cloud provider特有的load balancer对外提供服务，cloud provider负责将load balancer的流量导向Service。目前支持的cloud provider有GCP、AWS、Azur等。

下面我们来实践NodePort，Service httpd-svc的配置文件修改如图6-21所示。

```
apiVersion: v1
kind: Service
metadata:
  name: httpd-svc
spec:
  type: NodePort
  selector:
    run: httpd
  ports:
    - protocol: TCP
      port: 8080
      targetPort: 80
```

图6-21

添加type: NodePort，重新创建httpd-svc，如图6-22所示。

```
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl apply -f httpd-svc.yml
service "httpd-svc" created
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get service httpd-svc
NAME          CLUSTER-IP   EXTERNAL-IP   PORT(S)          AGE
httpd-svc     10.109.144.35 <nodes>       8080:32312/TCP   5s
ubuntu@k8s-master:~$
```

图6-22

Kubernetes依然会为httpd-svc分配一个ClusterIP，不同的是：

(1) EXTERNAL-IP为nodes，表示可通过Cluster每个节点自身的IP访问Service。

(2) PORT(S)为8080:32312。8080是ClusterIP监听的端口，32312则是节点上监听的端口。Kubernetes会从30000~32767中分配一个可用的端口，每个节点都会监听此端口并将请求转发给Service，如图6-23所示。

```
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ netstat -anl | grep 32312
tcp6      0      0 :::32312          :::*               LISTEN
ubuntu@k8s-master:~$
```

图6-23

下面测试NodePort是否正常工作，如图6-24所示。

```
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ curl 192.168.56.105:32312
<html><body><h1>It works!</h1></body></html>
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ curl 192.168.56.106:32312
<html><body><h1>It works!</h1></body></html>
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ curl 192.168.56.107:32312
<html><body><h1>It works!</h1></body></html>
ubuntu@k8s-master:~$
```

图6-24

通过三个节点IP+32312端口都能够访问httpd-svc。

接下来我们深入探讨一个问题：Kubernetes是如何将<NodeIP>:<NodePort>映射到Pod的呢？

与ClusterIP一样，也是借助了iptables。与ClusterIP相比，每个节点的iptables中都增加了下面两条规则，如图6-25所示。

```
-A KUBE-NODEPORTS -p tcp -m comment --comment "default/httpd-svc: " -m tcp --dport 32312 -j KUBE-MARK-MASQ
-A KUBE-NODEPORTS -p tcp -m comment --comment "default/httpd-svc: " -m tcp --dport 32312 -j KUBE-SVC-RL3JAE4GN7VOGDGP
```

图6-25

规则的含义是：访问当前节点32312端口的请求会应用规则KUBE-SVC-RL3JAE4GN7VOGDGP，内容如图6-26所示。

```
-A KUBE-SVC-RL3JAE4GN7VOGDGP -m comment --comment "default/httpd-svc: " -m statistic --mode random --probability 0.33332999982 -j KUBE-SEP-C5KB52P4BBJQ35PH
-A KUBE-SVC-RL3JAE4GN7VOGDGP -m comment --comment "default/httpd-svc: " -m statistic --mode random --probability 0.50000000000 -j KUBE-SEP-HGVKQZCF7RV4IT
-A KUBE-SVC-RL3JAE4GN7VOGDGP -m comment --comment "default/httpd-svc: " -j KUBE-SEP-XE25WGVXLHEIRV05
```

图6-26

其作用就是负载均衡到每一个Pod。

NodePort默认的是随机选择，不过我们可以用nodePort指定某个特定端口，如图6-27所示。

```
apiVersion: v1
kind: Service
metadata:
  name: httpd-svc
spec:
  type: NodePort
  selector:
    run: httpd
  ports:
  - protocol: TCP
    nodePort: 30000
    port: 8080
    targetPort: 80
```

图6-27

现在配置文件中就有三个Port了：

- nodePort是节点上监听的端口。
- port是ClusterIP上监听的端口。
- targetPort是Pod监听的端口。

最终，Node和ClusterIP在各自端口上接收到的请求都会通过iptables转发到Pod的targetPort。

应用新的nodePort并验证，如图6-28所示。

```
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl apply -f httpd-svc.yml
service "httpd-svc" configured
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get service httpd-svc
NAME          CLUSTER-IP      EXTERNAL-IP      PORT(S)          AGE
httpd-svc     10.109.144.35   <nodes>          8080:30000/TCP   36m
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ curl 192.168.56.105:30000
<html><body><h1>It works!</h1></body></html>
ubuntu@k8s-master:~$ curl 192.168.56.106:30000
<html><body><h1>It works!</h1></body></html>
ubuntu@k8s-master:~$ curl 192.168.56.107:30000
<html><body><h1>It works!</h1></body></html>
ubuntu@k8s-master:~$
```

图6-28

nodePort: 30000已经生效了。

6.5 小结

本章我们讨论访问应用的机制Service，学习了如何创建Service，Service的三种类型ClusterIP、NodePort和LoadBalancer，以及它们各自的适用场景。

第7章 Rolling Update

滚动更新是一次只更新一小部分副本，成功后再更新更多的副本，最终完成所有副本的更新。滚动更新的最大好处是零停机，整个更新过程始终有副本在运行，从而保证了业务的连续性。

7.1 实践

下面我们部署三副本应用，初始镜像为httpd:2.2.31，然后将其更新到httpd:2.2.32。

httpd:2.2.31的配置文件如图7-1所示。

```

apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: httpd
spec:
  replicas: 3
  template:
    metadata:
      labels:
        run: httpd
    spec:
      containers:
      - name: httpd
        image: :httpd:2.2.31
        ports:
        - containerPort: 80

```

图7-1

通过kubectl apply部署，如图7-2所示。

```

ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl apply -f httpd.yml
deployment "httpd" created
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get deployment httpd -o wide
NAME      DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE   CONTAINER(S)   IMAGE(S)           SELECTOR
httpd     3         3         3             3           8s    httpd           httpd:2.2.31       run=httpd
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get replicaset -o wide
NAME      DESIRED   CURRENT   READY   AGE   CONTAINER(S)   IMAGE(S)           SELECTOR
httpd-551879778  3         3         3       13s    httpd           httpd:2.2.31       pod-template-hash=55
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get pod
NAME      READY   STATUS    RESTARTS   AGE
httpd-551879778-jkxn4  1/1     Running   0          19s
httpd-551879778-n3sqv  1/1     Running   0          19s
httpd-551879778-zdfkt  1/1     Running   0          19s
ubuntu@k8s-master:~$

```

图7-2

部署过程如下：

- (1) 创建Deployment httpd。
- (2) 创建ReplicaSet httpd-551879778。
- (3) 创建三个Pod。

(4) 当前镜像为httpd:2.2.31。

将配置文件中的httpd:2.2.31替换为httpd:2.2.32，再次执行kubectl apply，如图7-3所示。

```
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ kubectl apply -f httpd.yml  
deployment "httpd" configured  
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ kubectl get deployment httpd -o wide  
NAME      DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE   CONTAINER(S)   IMAGE(S)           SELECTOR  
httpd     3         3         3            3           1m    httpd          httpd:2.2.32      run=httpd  
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ kubectl get replicaset -o wide  
NAME             DESIRED   CURRENT   READY   AGE   CONTAINER(S)   IMAGE(S)           SELECTOR  
httpd-1276601241 3         3         3       9s    httpd          httpd:2.2.32      pod-template-hash=1276601241  
httpd-551879778  0         0         0       2m    httpd          httpd:2.2.31      pod-template-hash=551879778  
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ kubectl get pod  
NAME                                READY   STATUS    RESTARTS   AGE  
httpd-1276601241-26jx3              1/1    Running   0          13s  
httpd-1276601241-27kh7              1/1    Running   0          10s  
httpd-1276601241-pwrt7              1/1    Running   0          11s  
ubuntu@k8s-master:~$
```

图7-3

我们发现了如下变化：

- (1) Deployment httpd的镜像更新为httpd:2.2.32。
- (2) 新创建了ReplicaSet httpd-1276601241，镜像为httpd:2.2.32，并且管理了三个新的Pod。
- (3) 之前的ReplicaSet httpd-551879778里面已经没有任何Pod。

结论是：ReplicaSet httpd-551879778的三个httpd:2.2.31 Pod已经被ReplicaSet httpd-1276601241的三个httpd:2.2.32 Pod替换了。

具体过程可以通过kubectl describe deployment httpd查看，如图7-4所示。

```
From: deployment-controller  
-----  
SubObjectPath: Type: Reason: Message:  
-----  
deployment-controller Normal ScalingReplicaSet Scaled up replica set httpd-551879778 to 3  
deployment-controller Normal ScalingReplicaSet Scaled up replica set httpd-1276601241 to 1  
deployment-controller Normal ScalingReplicaSet Scaled down replica set httpd-551879778 to 2  
deployment-controller Normal ScalingReplicaSet Scaled up replica set httpd-1276601241 to 2  
deployment-controller Normal ScalingReplicaSet Scaled down replica set httpd-551879778 to 1  
deployment-controller Normal ScalingReplicaSet Scaled up replica set httpd-1276601241 to 3  
deployment-controller Normal ScalingReplicaSet Scaled down replica set httpd-551879778 to 0
```

图7-4

每次只更新替换一个Pod：

- (1) ReplicaSet httpd-1276601241增加一个Pod，总数为1。
- (2) ReplicaSet httpd-551879778减少一个Pod，总数为2。
- (3) ReplicaSet httpd-1276601241增加一个Pod，总数为2。
- (4) ReplicaSet httpd-551879778减少一个Pod，总数为1。
- (5) ReplicaSet httpd-1276601241增加一个Pod，总数为3。
- (6) ReplicaSet httpd-551879778减少一个Pod，总数为0。

每次替换的Pod数量是可以定制的。Kubernetes提供了两个参数maxSurge和maxUnavailable来精细控制Pod的替换数量，我们将在后面结合Health Check特性一起讨论。

7.2 回滚

kubectl apply每次更新应用时，Kubernetes都会记录下当前的配置，保存为一个revision（版次），这样就可以回滚到某个特定revision。

默认配置下，Kubernetes只会保留最近的几个revision，可以在Deployment配置文件中通过revisionHistoryLimit属性增加revision数量。

下面实践回滚功能。应用有三个配置文件，即httpd.v1.yml、httpd.v2.yml和httpd.v3.yml，分别对应不同的httpd镜像2.4.16、2.4.17和2.4.18，如图7-5、图7-6、图7-7所示。

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: httpd
spec:
  revisionHistoryLimit: 10
  replicas: 3
  template:
    metadata:
      labels:
        run: httpd
    spec:
      containers:
      - name: httpd
        image: httpd:2.4.16
        ports:
        - containerPort: 80
```

图7-5

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: httpd
spec:
  revisionHistoryLimit: 10
  replicas: 3
  template:
    metadata:
      labels:
        run: httpd
    spec:
      containers:
      - name: httpd
        image: httpd:2.4.17
        ports:
        - containerPort: 80
```

图7-6

```

apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: httpd
spec:
  revisionHistoryLimit: 10
  replicas: 3
  template:
    metadata:
      labels:
        run: httpd
    spec:
      containers:
      - name: httpd
        image: httpd:2.4.18
        ports:
        - containerPort: 80

```

图7-7

通过kubectl apply部署并更新应用，如图7-8所示。

```

ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl apply -f httpd.v1.yml --record
deployment "httpd" created
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get deployment httpd -o wide
NAME      DESIRED  CURRENT  UP-TO-DATE  AVAILABLE  AGE      CONTAINER(S)  IMAGE(S)           SELECTOR
httpd     3        3        3           3          8s      httpd          httpd:2.4.16       run=httpd
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl apply -f httpd.v2.yml --record
deployment "httpd" configured
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get deployment httpd -o wide
NAME      DESIRED  CURRENT  UP-TO-DATE  AVAILABLE  AGE      CONTAINER(S)  IMAGE(S)           SELECTOR
httpd     3        3        3           3          27s     httpd          httpd:2.4.17       run=httpd
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl apply -f httpd.v3.yml --record
deployment "httpd" configured
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get deployment httpd -o wide
NAME      DESIRED  CURRENT  UP-TO-DATE  AVAILABLE  AGE      CONTAINER(S)  IMAGE(S)           SELECTOR
httpd     3        3        3           3          51s     httpd          httpd:2.4.18       run=httpd
ubuntu@k8s-master:~$

```

图7-8

--record的作用是将当前命令记录到revision记录中，这样我们就可以知道每个revision对应的是哪个配置文件了。通过kubectl rollout history deployment httpd查看revision历史记录，如图7-9所示。

```

ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl rollout history deployment httpd
deployments "httpd"
REVISION      CHANGE-CAUSE
1             kubectl apply --filename=httpd.v1.yml --record=true
2             kubectl apply --filename=httpd.v2.yml --record=true
3             kubectl apply --filename=httpd.v3.yml --record=true
ubuntu@k8s-master:~$

```

图7-9

CHANGE-CAUSE就是--record的结果。如果要回滚到某个版本，比如revision 1，可以执行命令kubectl rollout undo deployment httpd --to-revision=1，如图7-10所示。

```

ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl rollout undo deployment httpd --to-revision=1
deployment "httpd" rolled back
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get deployment httpd -o wide
NAME      DESIRED  CURRENT  UP-TO-DATE  AVAILABLE  AGE    CONTAINER(S)  IMAGE(S)           SELECTOR
httpd     3        3        3           3          8m    httpd         httpd:2.4.16      run=httpd
ubuntu@k8s-master:~$

```

图7-10

此时，revision历史记录也会发生相应变化，如图7-11所示。

```

ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl rollout history deployment httpd
deployments "httpd"
REVISION      CHANGE-CAUSE
2             kubectl apply --filename=httpd.v2.yml --record=true
3             kubectl apply --filename=httpd.v3.yml --record=true
4             kubectl apply --filename=httpd.v1.yml --record=true
ubuntu@k8s-master:~$

```

图7-11

revision 1变成了revision 4。不过我们可以通过CHANGE-CAUSE知道每个revision的具体含义，所以一定要在执行kubectl apply时加上--record参数。

7.3 小结

本章我们学习了滚动更新。滚动更新采用渐进的方式逐步替换旧版本Pod。如果更新不如预期，可以通过回滚操作恢复到更新前的状态。

第8章 Health Check

强大的自愈能力是Kubernetes这类容器编排引擎的一个重要特性。自愈的默认实现方式是自动重启发生故障的容器。除此之外，用户还可以利用Liveness和Readiness探测机制设置更精细的健康检查，进而实现如下需求：

- (1) 零停机部署。
- (2) 避免部署无效的镜像。
- (3) 更加安全的滚动升级。

下面通过实践学习Kubernetes的Health Check功能。

8.1 默认的健康检查

我们首先学习Kubernetes默认的健康检查机制：每个容器启动时都会执行一个进程，此进程由Dockerfile的CMD或ENTRYPOINT指定。如果进程退出时返回码非零，则认为容器发生故障，Kubernetes就会根据restartPolicy重启容器。

下面我们模拟一个容器发生故障的场景，Pod配置文件如图8-1所示。

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: healthcheck
  name: healthcheck
spec:
  restartPolicy: OnFailure
  containers:
  - name: healthcheck
    image: busybox
    args:
    - /bin/sh
    - -c
    - sleep 10; exit 1
```

图8-1

Pod的restartPolicy设置为OnFailure，默认为Always。

sleep 10; exit 1模拟容器启动10秒后发生故障。

执行kubectl apply创建Pod，命名为healthcheck，如图8-2所示。

```
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ kubectl apply -f healthcheck.yml  
pod "healthcheck" created  
ubuntu@k8s-master:~$
```

图8-2

过几分钟查看Pod的状态，如图8-3所示。

```
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ kubectl get pod healthcheck  
NAME          READY   STATUS    RESTARTS   AGE  
healthcheck   1/1     Running   3           1m  
ubuntu@k8s-master:~$
```

图8-3

可看到容器当前已经重启了3次。

在上面的例子中，容器进程返回值非零，Kubernetes则认为容器发生故障，需要重启。有不少情况是发生了故障，但进程并不会退出。比如访问Web服务器时显示500内部错误，可能是系统超载，也可能是资源死锁，此时httpd进程并没有异常退出，在这种情况下重启容器可能是最直接、最有效的解决方案，那我们如何利用Health Check机制来处理这类场景呢？

答案就是Liveness探测。

8.2 Liveness探测

Liveness探测让用户可以自定义判断容器是否健康的条件。如果探测失败，Kubernetes就会重启容器。

下面举例说明，创建Pod，如图8-4所示。

```

apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness
spec:
  restartPolicy: OnFailure
  containers:
  - name: liveness
    image: busybox
    args:
    - /bin/sh
    - -c
    - touch /tmp/healthy; sleep 30; rm -rf /tmp/healthy; sleep 600
    livenessProbe:
      exec:
        command:
        - cat
        - /tmp/healthy
      initialDelaySeconds: 10
      periodSeconds: 5

```

图8-4

启动进程首先创建文件/tmp/healthy，30秒后删除，在我们的设定中，如果/tmp/healthy文件存在，则认为容器处于正常状态，反之则发生故障。

livenessProbe部分定义如何执行Liveness探测：

（1）探测的方法是：通过cat命令检查/tmp/healthy文件是否存在。如果命令执行成功，返回值为零，Kubernetes则认为本次Liveness探测成功；如果命令返回值非零，本次Liveness探测失败。

（2）**initialDelaySeconds**: 10指定容器启动10之后开始执行Liveness探测，我们一般会根据应用启动的准备时间来设置。比如某个应用正常启动要花30秒，那么**initialDelaySeconds**的值就应该大于30。

（3）**periodSeconds**: 5指定每5秒执行一次Liveness探测。Kubernetes如果连续执行3次Liveness探测均失败，则会杀掉并重启容器。

下面创建Pod liveness，如图8-5所示。

```

ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl apply -f liveness.yml
pod "liveness" created
ubuntu@k8s-master:~$

```

图8-5

从配置文件可知，最开始的30秒，`/tmp/healthy`存在，`cat`命令返回0，`Liveness`探测成功，这段时间`kubectl describe pod liveness`的Events部分会显示正常的日志，如图8-6所示。

Type	Reason	Message
Normal	Scheduled	Successfully assigned liveness to k8s-node1
Normal	SuccessfulMountVolume	MountVolume.SetUp succeeded for volume "default-token-hnz7b"
Normal	Pulling	pulling image "busybox"
Normal	Pulled	Successfully pulled image "busybox"
Normal	Created	Created container
Normal	Started	Started container

图8-6

35秒之后，日志会显示`/tmp/healthy`已经不存在，`Liveness`探测失败。再过几十秒，几次探测都失败后，容器会被重启，如图8-7、图8-8所示。

Type	Reason	Message
Normal	Scheduled	Successfully assigned liveness to k8s-node1
Normal	SuccessfulMountVolume	MountVolume.SetUp succeeded for volume "default-token-hnz7b"
Warning	Unhealthy	Liveness probe failed: cat: can't open '/tmp/healthy': No such file
Normal	Pulling	pulling image "busybox"
Normal	Killing	Killing container with id docker://liveness:pod "liveness_default(308650ed-a32f-11e7-b2d9-0800274451ad)" container "liveness" is unhealthy, it will be killed and re-created.
Normal	Pulled	Successfully pulled image "busybox"
Normal	Created	Created container
Normal	Started	Started container

图8-7

```
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get pod liveness
NAME      READY   STATUS    RESTARTS   AGE
liveness  1/1     Running   1          1m
ubuntu@k8s-master:~$
```

图8-8

8.3 Readiness探测

除了`Liveness`探测，Kubernetes Health Check机制还包括`Readiness`探测。

用户通过`Liveness`探测可以告诉Kubernetes什么时候通过重启容器实现自愈；`Readiness`探测则是告诉Kubernetes什么时候可以将容器加入到

Service负载均衡池中，对外提供服务。

Readiness探测的配置语法与Liveness探测完全一样，如图8-9中的例子所示。

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: readiness
  name: readiness
spec:
  restartPolicy: OnFailure
  containers:
  - name: readiness
    image: busybox
    args:
    - /bin/sh
    - -c
    - touch /tmp/healthy; sleep 30; rm -rf /tmp/healthy; sleep 600
    readinessProbe:
      exec:
        command:
        - cat
        - /tmp/healthy
      initialDelaySeconds: 10
      periodSeconds: 5
```

图8-9

这个配置文件只是将前面例子中的liveness替换为了readiness，我们看看有什么不同的效果，如图8-10所示。

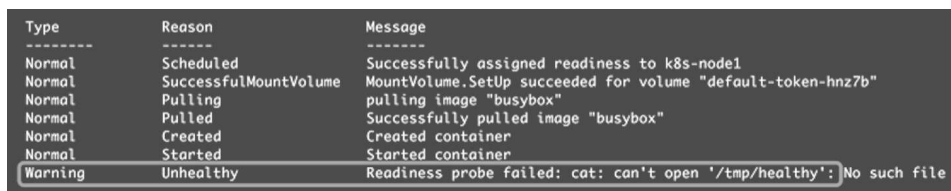
```
ubuntu@k8s-master:~$ kubectl apply -f readiness.y
pod "readiness" created
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get pod readiness
NAME      READY   STATUS    RESTARTS   AGE
readiness 0/1     Running   0           9s
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get pod readiness
NAME      READY   STATUS    RESTARTS   AGE
readiness 1/1     Running   0          16s
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get pod readiness
NAME      READY   STATUS    RESTARTS   AGE
readiness 0/1     Running   0          53s
ubuntu@k8s-master:~$
```

图8-10

Pod readiness的READY状态经历了如下变化：

- (1) 刚被创建时，READY状态为不可用。
- (2) 15秒后（`initialDelaySeconds + periodSeconds`），第一次进行Readiness探测并成功返回，设置READY为可用。
- (3) 30秒后，`/tmp/healthy`被删除，连续3次Readiness探测均失败后，READY被设置为不可用。

通过`kubectl describe pod readiness`也可以看到Readiness探测失败的日志，如图8-11所示。



Type	Reason	Message
Normal	Scheduled	Successfully assigned readiness to k8s-node1
Normal	SuccessfulMountVolume	MountVolume.SetUp succeeded for volume "default-token-hnz7b"
Normal	Pulling	pulling image "busybox"
Normal	Pulled	Successfully pulled image "busybox"
Normal	Created	Created container
Normal	Started	Started container
Warning	Unhealthy	Readiness probe failed: cat: can't open '/tmp/healthy': No such file

图8-11

下面对Liveness探测和Readiness探测做个比较：

- (1) Liveness探测和Readiness探测是两种Health Check机制，如果不特意配置，Kubernetes将对两种探测采取相同的默认行为，即通过判断容器启动进程的返回值是否为零来判断探测是否成功。
- (2) 两种探测的配置方法完全一样，支持的配置参数也一样。不同之处在于探测失败后的行为：Liveness探测是重启容器；Readiness探测则是将容器设置为不可用，不接收Service转发的请求。
- (3) Liveness探测和Readiness探测是独立执行的，二者之间没有依赖，所以可以单独使用，也可以同时使用。用Liveness探测判断容器是否需要重启以实现自愈；用Readiness探测判断容器是否已经准备好对外提供服务。

理解了Liveness探测和Readiness探测的原理，接下来讨论如何在业务场景中使用Health Check。

8.4 Health Check在Scale Up中的应用

对于多副本应用，当执行Scale Up操作时，新副本会作为backend被添加到Service的负载均衡中，与已有副本一起处理客户的请求。考虑到应用启动通常都需要一个准备阶段，比如加载缓存数据、连接数据库等，从容器启动到真正能够提供服务是需要一段时间的。我们可以通过Readiness探测判断容器是否就绪，避免将请求发送到还没有准备好的backend。

示例应用的配置文件如图8-12所示。

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: web
spec:
  replicas: 3
  template:
    metadata:
      labels:
        run: web
    spec:
      containers:
        - name: web
          image: myhttpd
          ports:
            - containerPort: 8080
            readinessProbe:
              httpGet:
                scheme: HTTP
                path: /healthy
                port: 8080
              initialDelaySeconds: 10
              periodSeconds: 5
---
apiVersion: v1
kind: Service
metadata:
  name: web-svc
spec:
  selector:
    run: web
  ports:
    - protocol: TCP
      port: 8080
      targetPort: 80
```

图8-12

重点关注readinessProbe部分。这里我们使用了不同于exec的另一种探测方法httpGet。Kubernetes对于该方法探测成功的判断条件是http请求的返回代码在200~400之间。

- schema指定协议，支持HTTP（默认值）和HTTPS。
- path指定访问路径。
- port指定端口。

上面配置的作用是：

- （1）容器启动10秒之后开始探测。
- （2）如果http://[container_ip]:8080/healthy返回代码不是200~400，表示容器没有就绪，不接收Service web-svc的请求。
- （3）每隔5秒探测一次。
- （4）直到返回代码为200~400，表明容器已经就绪，然后将其加入到web-svc的负载均衡中，开始处理客户请求。
- （5）探测会继续以5秒的间隔执行，如果连续发生3次失败，容器又会从负载均衡中移除，直到下次探测成功重新加入。

对于http://[container_ip]:8080/healthy，应用则可以实现自己的判断逻辑，比如检查所依赖的数据库是否就绪，示例代码如图8-13所示。

```

http.HandleFunc("/healthy", func(w http.ResponseWriter, r *http.Request) { ①

    healthy = True;

    // Check Database
    db = connect(dbIP, dbPort, dbUser, dbPassword) ②

    if db != NULL {
        try {
            db.Query("SELECT test;")
        } catch (e){
            err = e.message
        }
    }

    if db == NULL || err != NULL {
        healthy = False
        errMsg += "Database is not ready."
    }

    if healthy {
        w.Write([]byte("OK")) ③
    } else {
        // Send 503
        http.Error(w, errMsg, http.StatusServiceUnavailable) ④
    }
})

http.ListenAndServe(":8080") ⑤

```

图8-13

- ① 定义/healthy的处理函数。
- ② 连接数据库并执行测试SQL。
- ③ 测试成功，正常返回，代码200。
- ④ 测试失败，返回错误代码503。
- ⑤ 在8080端口监听。

对于生产环境中重要的应用，都建议配置Health Check，保证处理客户请求的容器都是准备就绪的Service backend。

8.5 Health Check在滚动更新中的应用

Health Check另一个重要的应用场景是**Rolling Update**。试想一下，现有一个正常运行的多副本应用，接下来对应用进行更新（比如使用更高版本的**image**），**Kubernetes**会启动新副本，然后发生了如下事件：

（1）正常情况下新副本需要**10**秒钟完成准备工作，在此之前无法响应业务请求。

（2）由于人为配置错误，副本始终无法完成准备工作（比如无法连接后端数据库）。

先别继续往下看，现在请花一分钟思考这个问题：如果没有配置**Health Check**，会出现怎样的情况？

因为新副本本身没有异常退出，默认的**Health Check**机制会认为容器已经就绪，进而会逐步用新副本替换现有副本，其结果就是：当所有旧副本都被替换后，整个应用将无法处理请求，无法对外提供服务。如果这是发生在重要的生产系统上，后果会非常严重。

如果正确配置了**Health Check**，新副本只有通过了**Readiness**探测才会被添加到**Service**；如果没有通过探测，现有副本不会被全部替换，业务仍然正常进行。

下面通过例子来实践**Health Check**在**Rolling Update**中的应用。

使用如下配置文件**app.v1.yml**模拟一个**10**副本的应用，如图**8-14**所示。

```

apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: app
spec:
  replicas: 10
  template:
    metadata:
      labels:
        run: app
    spec:
      containers:
      - name: app
        image: busybox
        args:
        - /bin/sh
        - -c
        - sleep 10; touch /tmp/healthy; sleep 30000
        readinessProbe:
          exec:
            command:
            - cat
            - /tmp/healthy
          initialDelaySeconds: 10
          periodSeconds: 5

```

图8-14

10秒后副本能够通过Readiness探测，如图8-15所示。

```

ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl apply -f app.v1.yml --record
deployment "app" created
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get deployment app
NAME          DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
app           10        10        10           10          28s
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get pod
NAME                                READY   STATUS    RESTARTS   AGE
app-2780995820-0mpfl               1/1     Running   0           32s
app-2780995820-9nmfm               1/1     Running   0           32s
app-2780995820-dqdw                1/1     Running   0           32s
app-2780995820-g0srs               1/1     Running   0           32s
app-2780995820-g52wp               1/1     Running   0           32s
app-2780995820-kddms               1/1     Running   0           32s
app-2780995820-rrwsh               1/1     Running   0           32s
app-2780995820-t3kl4               1/1     Running   0           32s
app-2780995820-v1qzn               1/1     Running   0           32s
app-2780995820-z8qx4               1/1     Running   0           32s
ubuntu@k8s-master:~$

```

图8-15

接下来滚动更新应用，配置文件app.v2.yml，如图8-16所示。

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: app
spec:
  replicas: 10
  template:
    metadata:
      labels:
        run: app
    spec:
      containers:
        - name: app
          image: busybox
          args:
            - /bin/sh
            - -c
            - sleep 3000
          readinessProbe:
            exec:
              command:
                - cat
                - /tmp/healthy
            initialDelaySeconds: 10
            periodSeconds: 5
```

图8-16

很显然，由于新副本中不存在/tmp/healthy，因此是无法通过Readiness探测的，验证如图8-17所示。

```

ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl apply -f app.v2.yml --record
deployment "app" configured
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get deployment app
NAME          DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
app           10        13        5            8           5m
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get pod
NAME                                READY   STATUS    RESTARTS   AGE
app-2780995820-0mpfl               1/1     Running   0          5m
app-2780995820-g0srs               1/1     Running   0          5m
app-2780995820-g52wp               1/1     Running   0          5m
app-2780995820-kddms               1/1     Running   0          5m
app-2780995820-rrwsh               1/1     Running   0          5m
app-2780995820-t3kl4               1/1     Running   0          5m
app-2780995820-v1qzn               1/1     Running   0          5m
app-2780995820-z8qx4               1/1     Running   0          5m
app-3350497563-d3ls3               0/1     Running   0          49s
app-3350497563-fkjvq               0/1     Running   0          49s
app-3350497563-ltjp3               0/1     Running   0          49s
app-3350497563-qm92c               0/1     Running   0          49s
app-3350497563-vh56z               0/1     Running   0          49s
ubuntu@k8s-master:~$

```

图8-17

这个截图包含了大量的信息，值得我们详细分析。

先关注kubectl get pod输出：

（1）从Pod的AGE栏可判断，最后5个Pod是新副本，目前处于NOT READY状态。

（2）旧副本从最初10个减少到8个。

再来看kubectl get deployment app的输出：

（1）DESIRED 10表示期望的状态是10个READY的副本。

（2）CURRENT 13表示当前副本的总数，即8个旧副本+5个新副本。

（3）UP-TO-DATE 5表示当前已经完成更新的副本数，即5个新副本。

(4) **AVAILABLE 8**表示当前处于**READY**状态的副本数，即8个旧副本。

在我们的设定中，新副本始终都无法通过**Readiness**探测，所以这个状态会一直保持下去。

上面我们模拟了一个滚动更新失败的场景。不过幸运的是：**Health Check**帮我们屏蔽了有缺陷的副本，同时保留了大部分旧副本，业务没有因更新失败受到影响。

接下来我们要回答：为什么新创建的副本数是5个，同时只销毁了2个旧副本？

原因是：滚动更新通过参数**maxSurge**和**maxUnavailable**来控制副本替换的数量。

1. **maxSurge**

此参数控制滚动更新过程中副本总数超过**DESIRED**的上限。**maxSurge**可以是具体的整数（比如3），也可以是百分百，向上取整。**maxSurge**默认值为25%。

在上面的例子中，**DESIRED**为10，那么副本总数的最大值为 $\text{roundUp}(10 + 10 * 25\%) = 13$ ，所以我们看到**CURRENT**就是13。

2. **maxUnavailable**

此参数控制滚动更新过程中，不可用的副本相占**DESIRED**的最大比例。**maxUnavailable**可以是具体的整数（比如3），也可以是百分百，向下取整。**maxUnavailable**默认值为25%。

在上面的例子中，**DESIRED**为10，那么可用的副本数至少要为 $10 - \text{roundDown}(10 * 25\%) = 8$ ，所以我们看到**AVAILABLE**是8。

maxSurge值越大，初始创建的新副本数量就越多；**maxUnavailable**值越大，初始销毁的旧副本数量就越多。

理想情况下，我们这个案例滚动更新的过程应该是这样的：

- (1) 创建3个新副本使副本总数达到13个。
- (2) 销毁2个旧副本使可用的副本数降到8个。
- (3) 当2个旧副本成功销毁后，再创建2个新副本，使副本总数保持为13个。
- (4) 当新副本通过Readiness探测后，会使可用副本数增加，超过8。
- (5) 进而可以继续销毁更多的旧副本，使可用副本数回到8。
- (6) 旧副本的销毁使副本总数低于13，这样就允许创建更多的新副本。
- (7) 这个过程会持续进行，最终所有的旧副本都会被新副本替换，滚动更新完成。

而我们的实际情况是在第4步就卡住了，新副本无法通过Readiness探测。这个过程可以在`kubectl describe deployment app`的日志部分查看，如图8-18所示。

Type	Reason	Message	
-----	-----	-----	
Normal	ScalingReplicaSet	Scaled up replica set app-2780995820 to 10	1
Normal	ScalingReplicaSet	Scaled up replica set app-3350497563 to 3	2
Normal	ScalingReplicaSet	Scaled down replica set app-2780995820 to 8	3
Normal	ScalingReplicaSet	Scaled up replica set app-3350497563 to 5	4

图8-18

如果滚动更新失败，可以通过`kubectl rollout undo`回滚到上一个版本，如图8-19所示。

```

ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl rollout history deployment app
deployments "app"
REVISION      CHANGE-CAUSE
1             kubectl apply --filename=app.v1.yml --record=true
2             kubectl apply --filename=app.v2.yml --record=true

ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl rollout undo deployment app --to-revision=1
deployment "app" rolled back
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get deployment app
NAME          DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
app           10        10        10           10          1h
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get pod
NAME                                READY   STATUS    RESTARTS   AGE
app-2780995820-0mpfl               1/1     Running   0           1h
app-2780995820-bq3zd               1/1     Running   0           1m
app-2780995820-g0srs               1/1     Running   0           1h
app-2780995820-g52wp               1/1     Running   0           1h
app-2780995820-kddms               1/1     Running   0           1h
app-2780995820-m0cvr               1/1     Running   0           1m
app-2780995820-rrwsh               1/1     Running   0           1h
app-2780995820-t3kl4               1/1     Running   0           1h
app-2780995820-v1qzn               1/1     Running   0           1h
app-2780995820-z8qx4               1/1     Running   0           1h
ubuntu@k8s-master:~$

```

图8-19

如果要定制maxSurge和maxUnavailable，可以进行如图8-20所示的配置。

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: app
spec:
  strategy:
    rollingUpdate:
      maxSurge: 35%
      maxUnavailable: 35%
  replicas: 10
  template:
    metadata:
      labels:
        run: app
    spec:
      containers:
        - name: app
          image: busybox
          args:
            - /bin/sh
            - -c
            - sleep 3000
          readinessProbe:
            exec:
              command:
                - cat
                - /tmp/healthy
            initialDelaySeconds: 10
            periodSeconds: 5
```

图8-20

8.6 小结

本章我们讨论了Kubernetes健康检查的两种机制：Liveness探测和Readiness探测，并实践了健康检查在Scale Up和Rolling Update场景中的应用。

第9章 数据管理

本章将讨论Kubernetes如何管理存储资源。

首先我们会学习Volume，以及Kubernetes如何通过Volume为集群中的容器提供存储；然后我们会实践几种常用的Volume类型并理解它们各自的应用场景；最后，我们会讨论Kubernetes如何通过Persistent Volume和Persistent Volume Claim分离集群管理员与集群用户的职责，并实践Volume的静态供给和动态供给。

9.1 Volume

本节我们讨论Kubernetes的存储模型Volume，学习如何将各种持久化存储映射到容器。

我们经常会说：容器和Pod是短暂的。其含义是它们的生命周期可能很短，会被频繁地销毁和创建。容器销毁时，保存在容器内部文件系统中的数据都会被清除。

为了持久化保存容器的数据，可以使用Kubernetes Volume。

Volume的生命周期独立于容器，Pod中的容器可能被销毁和重建，但Volume会被保留。

本质上，Kubernetes Volume是一个目录，这一点与Docker Volume类似。当Volume被mount到Pod，Pod中的所有容器都可以访问这个Volume。Kubernetes Volume也支持多种backend类型，包括emptyDir、hostPath、GCE Persistent Disk、AWS Elastic Block Store、NFS、Ceph等，完整列表可参考<https://kubernetes.io/docs/concepts/storage/volumes/#types-of-volumes>。

Volume提供了对各种backend的抽象，容器在使用Volume读写数据的时候不需要关心数据到底是存放在本地节点的文件系统中还是云硬盘上。对它来说，所有类型的Volume都只是一个目录。

我们将从最简单的emptyDir开始学习Kubernetes Volume。

9.1.1 emptyDir

emptyDir是最基础的Volume类型。正如其名字所示，一个emptyDir Volume是Host上的一个空目录。

emptyDir Volume对于容器来说是持久的，对于Pod则不是。当Pod从节点删除时，Volume的内容也会被删除。但如果只是容器被销毁而Pod还在，则Volume不受影响。

也就是说：emptyDir Volume的生命周期与Pod一致。

Pod中的所有容器都可以共享Volume，它们可以指定各自的mount路径。下面通过例子来实践emptyDir，配置文件如图9-1所示。

```
apiVersion: v1
kind: Pod
metadata:
  name: producer-consumer
spec:
  containers:
    - image: busybox
      name: producer
      volumeMounts:
        - mountPath: /producer_dir ②
          name: shared-volume
      args:
        - /bin/sh ③
        - -c
        - echo "hello world" > /producer_dir/hello ; sleep 30000

    - image: busybox
      name: consumer
      volumeMounts:
        - mountPath: /consumer_dir ④
          name: shared-volume
      args:
        - /bin/sh ⑤
        - -c
        - cat /consumer_dir/hello ; sleep 30000

  volumes:
    - name: shared-volume ①
      emptyDir: {}
```

图9-1

这里我们模拟了一个producer-consumer场景。Pod有两个容器producer和consumer，它们共享一个Volume。producer负责往Volume中写数据，consumer则是从Volume读取数据。

① 文件最底部volumes定义了一个emptyDir类型的Volume shared-volume。

② producer容器将shared-volume mount到/producer_dir目录。

③ producer通过echo将数据写到文件hello里。

④ consumer容器将shared-volume mount到/consumer_dir目录。

⑤ consumer通过cat从文件hello读数据。

执行命令创建Pod，如图9-2所示。

```
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ kubectl apply -f emptyDir.yml  
  
pod "producer-consumer" created  
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ kubectl get pod  
NAME                READY    STATUS    RESTARTS   AGE  
producer-consumer   2/2      Running   0           15s  
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ kubectl logs producer-consumer consumer  
hello world
```

图9-2

kubectl logs显示容器consumer成功读到了producer写入的数据，验证了两个容器共享emptyDir Volume。

因为emptyDir是Docker Host文件系统里的目录，其效果相当于执行了docker run -v/producer_dir和docker run -v /consumer_dir。通过docker inspect查看容器的详细配置信息，我们发现两个容器都mount了同一个目录，如图9-3、图9-4所示。

```
"mounts": [  
  {  
    "Source": "/var/lib/kubelet/pods/3e6100eb-a97a-11e7-8f72-080027  
4451ad/volumes/kubernetes.io~empty-dir/shared-volume",  
    "Destination": "/producer_dir",  
    "Mode": "",  
    "RW": true,  
    "Propagation": "rprivate"  
  },  
]
```

图9-3

```
"mounts": [  
  {  
    "Source": "/var/lib/kubelet/pods/3e6100eb-a97a-11e7-8f72-080027  
4451ad/volumes/kubernetes.io~empty-dir/shared-volume",  
    "Destination": "/consumer_dir",  
    "Mode": "",  
    "RW": true,  
    "Propagation": "rprivate"  
  },  
]
```

图9-4

这里 `/var/lib/kubelet/pods/3e6100eb-a97a-11e7-8f72-0800274451ad/volumes/kubernetes.io ~ empty-dir/shared-volume` 就是 `emptyDir` 在 Host 上的真正路径。

`emptyDir` 是 Host 上创建的临时目录，其优点是能够方便地为 Pod 中的容器提供共享存储，不需要额外的配置。它不具备持久性，如果 Pod 不存在了，`emptyDir` 也就没有了。根据这个特性，`emptyDir` 特别适合 Pod 中的容器需要临时共享存储空间场景，比如前面的生产者消费者用例。

9.1.2 hostPath

`hostPath Volume` 的作用是将 Docker Host 文件系统中已经存在的目录 mount 给 Pod 的容器。大部分应用都不会使用 `hostPath Volume`，因为这实际上增加了 Pod 与节点的耦合，限制了 Pod 的使用。不过那些需要访问 Kubernetes 或 Docker 内部数据（配置文件和二进制库）的应用则需要使用 `hostPath`。

比如 `kube-apiserver` 和 `kube-controller-manager` 就是这样的应用，通过 `kubectl edit --namespace=kube-system pod kube-apiserver-k8s-master` 查看 `kube-apiserver Pod` 的配置，`Volume` 的相关部分如图 9-5 所示。

```
volumeMounts:
- mountPath: /etc/kubernetes
  name: k8s
  readOnly: true
- mountPath: /etc/ssl/certs
  name: certs
- mountPath: /etc/pki
  name: pki
dnsPolicy: ClusterFirst
hostNetwork: true
nodeName: k8s-master
restartPolicy: Always
schedulerName: default-scheduler
securityContext: {}
terminationGracePeriodSeconds: 30
tolerations:
- effect: NoExecute
  operator: Exists
volumes:
- hostPath:
    path: /etc/kubernetes
    name: k8s
- hostPath:
    path: /etc/ssl/certs
    name: certs
- hostPath:
    path: /etc/pki
    name: pki
```

图9-5

这里定义了三个hostPath：volume k8s、certs和pki，分别对应Host目录/etc/kubernetes、/etc/ssl/certs和/etc/pki。

如果Pod被销毁了，hostPath对应的目录还是会被保留，从这一点来看，hostPath的持久性比emptyDir强。不过一旦Host崩溃，hostPath也就无法访问了。

接下来我们将学习具备真正持久性的Volume。

9.1.3 外部Storage Provider

如果Kubernetes部署在诸如AWS、GCE、Azure等公有云上，可以直接使用云硬盘作为Volume。下面给出一个AWS Elastic Block Store的例子，如图9-6所示。

```

apiVersion: v1
kind: Pod
metadata:
  name: using-ebs
spec:
  containers:
    - image: busybox
      name: using-ebs
      volumeMounts:
        - mountPath: /test-ebs
          name: ebs-volume
  volumes:
    - name: ebs-volume
      # This AWS EBS volume must already exist.
      awsElasticBlockStore:
        volumeID: <volume-id>
        fsType: ext4

```

图9-6

要在Pod中使用ESB volume，必须先在AWS中创建，然后通过volume-id引用。其他云硬盘的使用方法可参考各公有云厂商的官方文档。

Kubernetes Volume也可以使用主流的分式存储，比如Ceph、GlusterFS等。下面给出一个Ceph的例子，如图9-7所示。

```

apiVersion: v1
kind: Pod
metadata:
  name: using-ceph
spec:
  containers:
    - image: busybox
      name: using-ceph
      volumeMounts:
        - name: ceph-volume
          mountPath: /test-ceph
  volumes:
    - name: ceph-volume
      cephfs:
        path: /some/path/in/side/cephfs
        monitors: "10.16.154.78:6789"
        secretFile: "/etc/ceph/admin.secret"

```

图9-7

Ceph文件系统的/some/path/in/side/cephfs目录被mount到容器路径/test-ceph。

相对于emptyDir和hostPath，这些Volume类型的最大特点就是不依赖Kubernetes。Volume的底层基础设施由独立的存储系统管理，与Kubernetes集群是分离的。数据被持久化后，即使整个Kubernetes崩溃也不会受损。

当然，运维这样的存储系统通常不是一项简单的工作，特别是对可靠性、可用性和扩展性有较高要求的时候。

9.2 PersistentVolume & PersistentVolumeClaim

Volume提供了非常好的数据持久化方案，不过在可管理性上还有不足。

拿前面的AWS EBS例子来说，要使用Volume，Pod必须事先知道如下信息：

- (1) 当前Volume来自AWS EBS。
- (2) EBS Volume已经提前创建，并且知道确切的volume-id。

Pod通常是由应用的开发人员维护，而Volume则通常是由存储系统的管理员维护。开发人员要获得上面的信息，要么询问管理员，要么自己就是管理员。

这样就带来一个管理上的问题：应用开发人员和系统管理员的职责耦合在一起了。如果系统规模较小或者对于开发环境，这样的情况还可以接受，当集群规模变大，特别是对于生产环境，考虑到效率和安全性，这就成了必须要解决的问题。

Kubernetes 给出的解决方案是 PersistentVolume 和 PersistentVolumeClaim。

PersistentVolume (PV) 是外部存储系统中的一块存储空间，由管理员创建和维护。与**Volume**一样，**PV**具有持久性，生命周期独立于**Pod**。

PersistentVolumeClaim (PVC) 是对**PV**的申请 (**Claim**)。PVC通常由普通用户创建和维护。需要为**Pod**分配存储资源时，用户可以创建一个**PVC**，指明存储资源的容量大小和访问模式（比如只读）等信息，**Kubernetes**会查找并提供满足条件的**PV**。

有了**PersistentVolumeClaim**，用户只需要告诉**Kubernetes**需要什么样的存储资源，而不必关心真正的空间从哪里分配、如何访问等底层细节信息。这些**Storage Provider**的底层信息交给管理员来处理，只有管理员才应该关心创建**PersistentVolume**的细节信息。

Kubernetes支持多种类型的**PersistentVolume**，比如**AWS EBS**、**Ceph**、**NFS** 等，完整列表请参考 <https://kubernetes.io/docs/concepts/storage/persistent-volumes/#types-of-persistent-volumes>。

下面我们用**NFS**来体会**PersistentVolume**的使用方法。

9.2.1 NFS PersistentVolume

作为准备工作，我们已经在**k8s-master**节点上搭建了一个**NFS**服务器，目录为**/nfsdata**，如图9-8所示。

```
root@k8s-master:~#  
root@k8s-master:~# showmount -e  
Export list for k8s-master:  
/nfsdata *
```

图9-8

下面创建一个**PV mypv1**，配置文件**nfs-pv1.yml**如图9-9所示。

```

apiVersion: v1
kind: PersistentVolume
metadata:
  name: mypv1
spec:
  capacity:
    storage: 1Gi ①
  accessModes:
    - ReadWriteOnce ②
  persistentVolumeReclaimPolicy: Recycle ③
  storageClassName: nfs ④
  nfs:
    path: /nfsdata/pv1 ⑤
    server: 192.168.56.105

```

图9-9

① capacity指定PV的容量为1GB。

② accessModes指定访问模式为ReadWriteOnce，支持的访问模式有3种：ReadWriteOnce表示PV能以read-write模式mount到单个节点，ReadOnlyMany表示PV能以read-only模式mount到多个节点，ReadWriteMany表示PV能以read-write模式mount到多个节点。

③ persistentVolumeReclaimPolicy指定当PV的回收策略为Recycle，支持的策略有3种：Retain表示需要管理员手工回收；Recycle表示清除PV中的数据，效果相当于执行rm -rf/thevolume/*；Delete表示删除Storage Provider上的对应存储资源，例如AWS EBS、GCE PD、Azure Disk、OpenStack Cinder Volume等。

④ storageClassName指定PV的class为nfs。相当于为PV设置了一个分类，PVC可以指定class申请相应class的PV。

⑤ 指定PV在NFS服务器上对应的目录。

创建mypv1，如图9-10所示。

```

ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl apply -f nfs-pv1.yml
persistentvolume "mypv1" created
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get pv

```

NAME	CAPACITY	ACCESSMODES	RECLAIMPOLICY	STATUS	CLAIM	STORAGECLASS	REASON	AGE
mypv1	1Gi	RWO	Recycle	Available		nfs		15s

```

ubuntu@k8s-master:~$

```

图9-10

STATUS为Available，表示mypv1就绪，可以被PVC申请。

接下来创建PVC mypvc1，配置文件nfs-pvc1.yml如图9-11所示。

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: mypvc1
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
  storageClassName: nfs
```

图9-11

PVC就很简单了，只需要指定PV的容量、访问模式和class即可。

创建mypvc1，如图9-12所示。

```
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl apply -f nfs-pvc1.yml
persistentvolumeclaim "mypvc1" created
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get pvc
NAME      STATUS   VOLUME   CAPACITY   ACCESSMODES   STORAGECLASS   AGE
mypvc1    Bound    mypv1    1Gi        RWO           nfs            22s
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get pv
NAME      CAPACITY   ACCESSMODES   RECLAIMPOLICY   STATUS   CLAIM          STORAGECLASS   REASON   AGE
mypv1     1Gi        RWO           Recycle         Bound    default/mypvc1  nfs          6m
```

图9-12

从kubectl get pvc和kubectl get pv的输出可以看到mypvc1已经Bound到mypv1，申请成功。

接下来就可以在Pod中使用存储了，Pod配置文件pod1.yml如图9-13所示。

```

kind: Pod
apiVersion: v1
metadata:
  name: mypod1
spec:
  containers:
    - name: mypod1
      image: busybox
      args:
        - /bin/sh
        - -c
        - sleep 30000
      volumeMounts:
        - mountPath: "/mydata"
          name: mydata
  volumes:
    - name: mydata
      persistentVolumeClaim:
        claimName: mypvc1

```

图9-13

与使用普通 Volume 的格式类似，在 volumes 中通过 persistentVolumeClaim 指定使用 mypvc1 申请的 Volume。

创建 mypod1，如图9-14所示。

```

ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl apply -f pod1.yml
pod "mypod1" created
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get pod -o wide

```

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
mypod1	1/1	Running	0	32s	10.244.4.60	k8s-node1

```

ubuntu@k8s-master:~$

```

图9-14

验证PV是否可用，如图9-15所示。

```

ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl exec mypod1 touch /mydata/hello
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ ls /nfsdata/pv1/
hello
ubuntu@k8s-master:~$

```

图9-15

可见，在Pod中创建的文件/mydata/hello确实已经保存到了NFS服务器目录/nfsdata/pv1中。

9.2.2 回收PV

当不需要使用PV时，可用删除PVC回收PV，如图9-16所示。

```
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ kubectl delete pvc mypvc1  
persistentvolumeclaim "mypvc1" deleted  
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ kubectl get pod -o wide  
NAME          READY   STATUS    RESTARTS   AGE   IP            NODE  
mypod1        1/1     Running   0           25m   10.244.4.60   k8s-node1  
recycler-for-mypv1 0/1     ContainerCreating 0       2s    <none>        k8s-node1  
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ kubectl get pv  
NAME          CAPACITY   ACCESSMODES   RECLAIMPOLICY   STATUS   CLAIM          STORAGECLASS   REA  
mypv1         1Gi        RWO           Recycle         Released default/mypvc1 nfs
```

图9-16

当PVC mypvc1被删除后，我们发现Kubernetes启动了一个新Pod recycler-for-mypv1，这个Pod的作用就是清除PV mypv1的数据。此时mypv1的状态为Released，表示已经解除了与mypvc1的Bound，正在清除数据，不过此时还不可用。

当数据清除完毕，mypv1的状态重新变为Available，此时可以被新的PVC申请，如图9-17所示。

```
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ kubectl get pv  
NAME          CAPACITY   ACCESSMODES   RECLAIMPOLICY   STATUS   CLAIM          STORAGECLASS  
mypv1         1Gi        RWO           Recycle         Available default/mypvc1 nfs  
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ ls /nfsdata/pv1/  
ubuntu@k8s-master:~$
```

图9-17

/nfsdata/pv1中的hello文件已经被删除了。

因为PV的回收策略设置为Recycle，所以数据会被清除，但这可能不是我们想要的结果。如果我们希望保留数据，可以将策略设置为Retain，如图9-18所示。

```

apiVersion: v1
kind: PersistentVolume
metadata:
  name: mypv1
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  storageClassName: nfs
  nfs:
    path: /nfsdata/pv1
    server: 192.168.56.105

```

图9-18

通过kubectl apply更新PV，如图9-19所示。

```

ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl apply -f nfs-pv1.yml
persistentvolume "mypv1" configured
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get pv

```

NAME	CAPACITY	ACCESSMODES	RECLAIMPOLICY	STATUS	CLAIM	STORAGECLASS
mypv1	1Gi	RWO	Retain	Available		nfs

```

ubuntu@k8s-master:~$

```

图9-19

回收策略已经变为Retain，通过下面的步骤验证其效果，如图9-20所示。

```

ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl apply -f nfs-pvc1.yml ①
persistentvolumeclaim "mypvc1" created
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl exec mypod1 touch /mydata/hello ②
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ ls /nfsdata/pv1/
hello
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl delete pvc mypvc1 ③
persistentvolumeclaim "mypvc1" deleted
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get pv

```

NAME	CAPACITY	ACCESSMODES	RECLAIMPOLICY	STATUS	CLAIM	STORAGECLASS
mypv1	1Gi	RWO	Retain	Released	default/mypvc1	nfs

```

ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get pod -o wide ④

```

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
mypod1	1/1	Running	0	46m	10.244.4.60	k8s-node1

```

ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ ls /nfsdata/pv1/ ⑤
hello
ubuntu@k8s-master:~$

```

图9-20

- ① 重新创建mypvc1。
- ② 在mypv1中创建文件hello。
- ③ mypv1状态变为Released。
- ④ Kubernetes并没有启动Pod recycler-for-mypv1。
- ⑤ PV中的数据被完整保留。

虽然mypv1中的数据得到了保留，但其PV状态会一直处于Released，不能被其他PVC申请。为了重新使用存储资源，可以删除并重新创建mypv1。删除操作只是删除了PV对象，存储空间中的数据并不会被删除。

新建的mypv1状态为Available，如图9-21所示，已经可以被PVC申请。

```
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl delete pv mypv1
persistentvolume "mypv1" deleted
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl apply -f nfs-pv1.yml
persistentvolume "mypv1" created
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get pv
NAME      CAPACITY  ACCESSMODES  RECLAIMPOLICY  STATUS    CLAIM  STORAGECLASS  REASON  AGE
mypv1     1Gi       RWO          Retain         Available    


```

图9-21

PV还支持Delete的回收策略，会删除PV在Storage Provider上对应的存储空间。NFS的PV不支持Delete，支持Delete的Provider有AWS EBS、GCE PD、Azure Disk、OpenStack Cinder Volume等。

9.2.3 PV动态供给

在前面的例子中，我们提前创建了PV，然后通过PVC申请PV并在Pod中使用，这种方式叫作静态供给（Static Provision）。

与之对应的是动态供给（Dynamical Provision），即如果没有满足PVC条件的PV，会动态创建PV。相比静态供给，动态供给有明显的优

势：不需要提前创建PV，减少了管理员的工作量，效率高。

动态供给是通过StorageClass实现的，StorageClass定义了如何创建PV，下面给出两个例子。

(1) StorageClass standard，如图9-22所示。

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: standard
provisioner: kubernetes.io/aws-ebs
parameters:
  type: gp2
reclaimPolicy: Retain
```

图9-22

(2) StorageClass slow，如图9-23所示。

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: slow
provisioner: kubernetes.io/aws-ebs
parameters:
  type: io1
  zones: us-east-1d, us-east-1c
  iopsPerGB: "10"
```

图9-23

这两个StorageClass都会动态创建AWS EBS，不同点在于standard创建的是gp2类型的EBS，而slow创建的是io1类型的EBS。不同类型的EBS支持的参数可参考AWS官方文档。

StorageClass支持Delete和Retain两种reclaimPolicy，默认是Delete。

与之前一样，PVC在申请PV时，只需要指定StorageClass、容量以及访问模式即可，如图9-24所示。

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: mypvc1
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
      storageClassName: standard
```

图9-24

除了 AWS EBS，Kubernetes 还支持其他多种动态供给 PV 的 Provisioner，完整列表请参考 <https://kubernetes.io/docs/concepts/storage/storage-classes/#provisioner>。

9.3 一个数据库例子

本节演示如何为MySQL数据库提供持久化存储，步骤为：

- (1) 创建PV和PVC。
- (2) 部署MySQL。
- (3) 向MySQL添加数据。
- (4) 模拟节点宕机故障，Kubernetes将MySQL自动迁移到其他节点。
- (5) 验证数据一致性。

首先创建PV和PVC，配置说明如下。

- mysql-pv.yml如图9-25所示。

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: mysql-pv
spec:
  accessModes:
    - ReadWriteOnce
  capacity:
    storage: 1Gi
  persistentVolumeReclaimPolicy: Retain
  storageClassName: nfs
  nfs:
    path: /nfsdata/mysql-pv
    server: 192.168.56.105
```

图9-25

- mysql-pvc.yml如图9-26所示。

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: mysql-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
  storageClassName: nfs
```

图9-26

创建mysql-pv和mysql-pvc，如图9-27所示。

```

ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl apply -f mysql-pv.yml
persistentvolume "mysql-pv" created
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl apply -f mysql-pvc.yml
persistentvolumeclaim "mysql-pvc" created
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get pv,pvc
NAME          CAPACITY  ACCESSMODES  RECLAIMPOLICY  STATUS  CLAIM                STORAGECLASS
pv/mysql-pv   1Gi       RWO          Retain         Bound   default/mysql-pvc   nfs

NAME          STATUS  VOLUME  CAPACITY  ACCESSMODES  STORAGECLASS  AGE
pvc/mysql-pvc Bound    mysql-pv  1Gi       RWO          nfs           9s
ubuntu@k8s-master:~$

```

图9-27

接下来部署MySQL，配置文件如图9-28所示。

```

apiVersion: v1
kind: Service
metadata:
  name: mysql
spec:
  ports:
    - port: 3306
  selector:
    app: mysql

---
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: mysql
spec:
  selector:
    matchLabels:
      app: mysql
  template:
    metadata:
      labels:
        app: mysql
    spec:
      containers:
        - image: mysql:5.6
          name: mysql
          env:
            - name: MYSQL_ROOT_PASSWORD
              value: password
          ports:
            - containerPort: 3306
              name: mysql
          volumeMounts:
            - name: mysql-persistent-storage
              mountPath: /var/lib/mysql
          volumes:
            - name: mysql-persistent-storage
              persistentVolumeClaim:
                claimName: mysql-pvc

```

图9-28

PVC mysql-pvc Bound的PV mysql-pv将被mount到MySQL的数据目录 var/lib/mysql，如图9-29所示。

```

ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl apply -f mysql.yml
service "mysql" created
deployment "mysql" created
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get pod -o wide
NAME                READY   STATUS    RESTARTS   AGE   IP            NODE
mysql-2150355289-p99h8 1/1     Running   0          15s   10.244.5.80   k8s-node2
ubuntu@k8s-master:~$

```

图9-29

MySQL被部署到k8s-node2，下面通过客户端访问Service mysql，如图9-30所示。

```

kubectl run -it --rm --image=mysql:5.6 --restart=Never mysql-
client -- mysql-h mysql -ppassword

```

```

ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl run -it --rm --image=mysql:5.6 --restart=Never mysql-client -- mysql -h mysql -ppassword
If you don't see a command prompt, try pressing enter.
mysql>

```

图9-30

更新数据库，如图9-31所示。

```

mysql>
mysql> use mysql ①
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> create table my_id( id int(4) ); ②
Query OK, 0 rows affected (0.01 sec)

mysql> insert my_id values( 111 ); ③
Query OK, 1 row affected (0.00 sec)

mysql> select * from my_id; ④
+-----+
| id |
+-----+
| 111 |
+-----+
1 row in set (0.00 sec)

mysql>

```

图9-31

① 切换到数据库mysql。

② 创建数据库表my_id。

③ 插入一条数据。

④ 确认数据已经写入。

关闭k8s-node2，模拟节点宕机故障，如图9-32所示。

```
root@k8s-node2:~#  
root@k8s-node2:~# shutdown now  
Connection to 192.168.56.107 closed by remote host.  
Connection to 192.168.56.107 closed.
```

图9-32

一段时间后，Kubernetes将MySQL迁移到k8s-node1，如图9-33所示。

```
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ kubectl get pod -o wide  
NAME                READY    STATUS    RESTARTS   AGE    IP             NODE  
mysql-2150355289-p13n5 1/1      Running   0          31s    10.244.4.66    k8s-node1  
mysql-2150355289-p99h8 1/1      Unknown   0          16m    10.244.5.80    k8s-node2  
ubuntu@k8s-master:~$
```

图9-33

验证数据的一致性，如图9-34所示。

```
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ kubectl run -it --rm --image=mysql:5.6 --restart=Never mysql-client -- mysql -h mysql -ppassword  
If you don't see a command prompt, try pressing enter.  
  
mysql> use mysql  
Reading table information for completion of table and column names  
You can turn off this feature to get a quicker startup with -A  
  
Database changed  
mysql> select * from my_id;  
+-----+  
| id |  
+-----+  
| 111 |  
+-----+  
1 row in set (0.00 sec)  
  
mysql>
```

图9-34

MySQL服务恢复，数据也完好无损。

9.4 小结

本章我们讨论了Kubernetes如何管理存储资源。

`emptyDir` 和 `hostPath` 类型的 `Volume` 很方便，但可持久性不强，Kubernetes支持多种外部存储系统的`Volume`。

`PV`和`PVC`分离了管理员和普通用户的职责，更适合生产环境。我们还学习了如何通过`StorageClass`实现更高效的动态供给。

最后，我们演示了如何在MySQL中使用`PersistentVolume`实现数据持久性。

第10章 Secret & Configmap

应用启动过程中可能需要一些敏感信息，比如访问数据库的用户名、密码或者密钥。将这些信息直接保存在容器镜像中显然不妥，Kubernetes提供的解决方案是`Secret`。

`Secret`会以密文的方式存储数据，避免了直接在配置文件中保存敏感信息。`Secret`会以`Volume`的形式被`mount`到`Pod`，容器可通过文件的方式使用`Secret`中的敏感数据；此外，容器也可以环境变量的方式使用这些数据。

`Secret`可通过命令行或YAML创建。比如希望`Secret`中包含如下信息：用户名`admin`、密码`123456`。

10.1 创建Secret

有四种方法创建`Secret`：

(1) 通过`--from-literal`：

```
kubectl create secret generic mysecret --from-literal=username=admin --from-literal=password=123456
```

每个`--from-literal`对应一个信息条目。

(2) 通过`--from-file`：

```
echo -n admin > ./username
```

```
echo -n 123456 > ./password
```

```
kubectl create secret generic mysecret --from-file=./username -  
-from-file=./password
```

每个文件内容对应一个信息条目。

(3) 通过--from-env-file:

```
cat << EOF > env.txt
```

```
username=admin
```

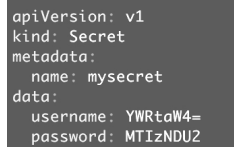
```
password=123456
```

```
EOF
```

```
kubectl create secret generic mysecret --from-env-  
file=env.txt
```

文件env.txt中每行Key=Value对应一个信息条目。

(4) 通过YAML配置文件，如图10-1所示。



```
apiVersion: v1  
kind: Secret  
metadata:  
  name: mysecret  
data:  
  username: YWRtaW4=  
  password: MTIzNDU2
```

图10-1

文件中的敏感数据必须是通过base64编码后的结果，如图10-2所示。

```
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ echo -n admin | base64  
YWRtaW4=  
ubuntu@k8s-master:~$ echo -n 123456 | base64  
MTIzNDU2  
ubuntu@k8s-master:~$
```

图10-2

执行kubectl apply创建Secret，如图10-3所示。

```
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ kubectl apply -f mysecrete.yml  
secret "mysecret" created  
ubuntu@k8s-master:~$
```

图10-3

10.2 查看Secret

通过kubectl get secret查看存在的secret，如图10-4所示。

```
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ kubectl get secret mysecret  
NAME          TYPE      DATA      AGE  
mysecret      Opaque    2          4m  
ubuntu@k8s-master:~$
```

图10-4

显示有两个数据条目，通过kubectl describe secret查看条目的Key，如图10-5所示。

```
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ kubectl describe secret mysecret  
Name:          mysecret  
Namespace:     default  
Labels:        <none>  
Annotations:   <none>  
  
Type:          Opaque  
  
Data  
====  
password:      6 bytes  
username:      5 bytes  
ubuntu@k8s-master:~$
```

图10-5

如果还想查看Value，可以用`kubectl edit secret mysecret`，如图10-6所示。

```
apiVersion: v1
data:
  password: MTIzNDU2
  username: YWRtaW4=
kind: Secret
metadata:
  creationTimestamp: 2017-10-10T07:16:21Z
  name: mysecret
  namespace: default
  resourceVersion: "1598871"
  selfLink: /api/v1/namespaces/default/secrets/mysecret
  uid: eaeb4980-ad8a-11e7-8f72-0800274451ad
type: Opaque
```

图10-6

然后通过base64将Value反编码，如图10-7所示。

```
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ echo -n MTIzNDU2 | base64 --decode
123456ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ echo -n YWRtaW4= | base64 --decode
adminubuntu@k8s-master:~$
ubuntu@k8s-master:~$
```

图10-7

10.3 在Pod中使用Secret

Pod可以通过Volume或者环境变量的方式使用Secret。

10.3.1 Volume方式

Pod的配置文件如图10-8所示。

```

apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: mypod
    image: busybox
    args:
      - /bin/sh
      - -c
      - sleep 10; touch /tmp/healthy; sleep 30000
    volumeMounts:
      - name: foo
        mountPath: "/etc/foo" ②
        readOnly: true
  volumes:
  - name: foo ①
    secret:
      secretName: mysecret

```

图10-8

① 定义volume foo，来源为secret mysecret。

② 将foo mount到容器路径/etc/foo，可指定读写权限为readOnly。

创建Pod并在容器中读取Secret，如图10-9所示。

```

ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl apply -f mypod.yml
pod "mypod" created
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl exec -it mypod sh
/ #
/ # ls /etc/foo
password username
/ #
/ # cat /etc/foo/username
admin/ #
/ # cat /etc/foo/password
123456/ #
/ #

```

图10-9

可以看到，Kubernetes会在指定的路径/etc/foo下为每条敏感数据创建一个文件，文件名就是数据条目的Key，这里是/etc/foo/username和/etc/foo/password，Value则以明文存放在文件中。

我们也可以自定义存放数据的文件名，比如将配置文件改为如图10-10所示那样。

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: mypod
    image: busybox
    args:
      - /bin/sh
      - -c
      - sleep 10; touch /tmp/healthy; sleep 30000
    volumeMounts:
      - name: foo
        mountPath: "/etc/foo"
        readOnly: true
  volumes:
  - name: foo
    secret:
      secretName: mysecret
      items:
        - key: username
          path: my-group/my-username
        - key: password
          path: my-group/my-password
```

图10-10

这时数据将分别存放在 `/etc/foo/my-group/my-username` 和 `/etc/foo/my-group/my-password` 中。

以 **Volume** 方式使用的 **Secret** 支持动态更新：Secret 更新后，容器中的数据也会更新。

将 `password` 更新为 `abcdef`，base64 编码为 `YWJjZGVm`，如图 10-11 所示。

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
data:
  username: YWRtaW4=
  password: YWJjZGVm
```

图10-11

更新Secret，如图10-12所示。

```
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ kubectl apply -f mysecrete.yml  
secret "mysecret" configured  
ubuntu@k8s-master:~$
```

图10-12

几秒钟后，新的password会同步到容器，如图10-13所示。

```
/ #  
/ # cat /etc/foo/password  
123456/ #  
/ #  
/ # cat /etc/foo/password  
abcdef/ #  
/ #
```

图10-13

10.3.2 环境变量方式

通过 Volume 使用 Secret，容器必须从文件读取数据，稍显麻烦，Kubernetes还支持通过环境变量使用Secret。

Pod配置文件示例如图10-14所示。

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: mypod
    image: busybox
    args:
      - /bin/sh
      - -c
      - sleep 10; touch /tmp/healthy; sleep 30000
    env:
      - name: SECRET_USERNAME
        valueFrom:
          secretKeyRef:
            name: mysecret
            key: username
      - name: SECRET_PASSWORD
        valueFrom:
          secretKeyRef:
            name: mysecret
            key: password
```

图10-14

创建Pod并读取Secret，如图10-15所示。

```
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl apply -f mypod-env.yml
pod "mypod" created
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl exec -it mypod sh
/ #
/ # echo $SECRET_USERNAME
admin
/ #
/ # echo $SECRET_PASSWORD
abcdef
/ #
```

图10-15

通过环境变量SECRET_USERNAME和SECRET_PASSWORD成功读取到Secret的数据。

需要注意的是，环境变量读取Secret很方便，但无法支撑Secret动态更新。

10.4 ConfigMap

Secret可以为Pod提供密码、Token、私钥等敏感数据；对于一些非敏感数据，比如应用的配置信息，则可以用ConfigMap。

ConfigMap的创建和使用方式与Secret非常类似，主要的不同是数据以明文的形式存放。

与Secret一样，ConfigMap也支持四种创建方式：

(1) 通过--from-literal:

```
kubectl create configmap myconfigmap --from-literal=config1=xxx--from-literal=config2=yyy
```

每个--from-literal对应一个信息条目。

(2) 通过--from-file:

```
echo -n xxx > ./config1
```

```
echo -n yyy > ./config2
```

```
kubectl create configmap myconfigmap --from-file=./config1 --from-file=./config2
```

每个文件内容对应一个信息条目。

(3) 通过--from-env-file:

```
cat << EOF > env.txt
```

```
config1=xxx
```

```
config2=yyy
```

```
EOF
```

```
kubectl create configmap myconfigmap --from-env-file=env.txt
```

文件env.txt中每行Key=Value对应一个信息条目。

(4) 通过YAML配置文件，如图10-16所示。文件中的数据直接以明文输入。

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: myconfigmap
data:
  config1: xxx
  config2: yyy
```

图10-16

与 Secret 一样，Pod 也可以通过 Volume 或者环境变量的方式使用 Secret。

(1) Volume方式如图10-17所示。

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: mypod
    image: busybox
    args:
      - /bin/sh
      - -c
      - sleep 10; touch /tmp/healthy; sleep 30000
    volumeMounts:
      - name: foo
        mountPath: "/etc/foo"
        readOnly: true
  volumes:
  - name: foo
    configMap:
      name: myconfigmap
```

图10-17

(2) 环境变量方式如图10-18所示。

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: mypod
    image: busybox
    args:
      - /bin/sh
      - -c
      - sleep 10; touch /tmp/healthy; sleep 30000
    env:
      - name: CONFIG_1
        valueFrom:
          configMapKeyRef:
            name: myconfigmap
            key: config1
      - name: CONFIG_2
        valueFrom:
          configMapKeyRef:
            name: myconfigmap
            key: config2
```

图10-18

大多数情况下，配置信息都以文件形式提供，所以在创建ConfigMap时通常采用--from-file或YAML方式，读取ConfigMap时通常采用Volume方式。比如给Pod传递如何记录日志的配置信息，如图10-19所示。

```
class: logging.handlers.RotatingFileHandler
formatter: precise
level: INFO
filename: %hostname-%timestamp.log
```

图10-19

可以采用--from-file形式，将其保存在文件logging.conf中，然后执行命令：

```
kubectl create configmap myconfigmap --from-file=./logging.conf
```

如果采用YAML配置文件，其内容则如图10-20所示。

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: myconfigmap
data:
  logging.conf: |
    class: logging.handlers.RotatingFileHandler
    formatter: precise
    level: INFO
    filename: %hostname-%timestamp.log
```

图10-20

注意，别漏写了Key logging.conf后面的|符号。

创建并查看ConfigMap，如图10-21所示。

```
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl apply -f myconfigmap.yml
configmap "myconfigmap" created
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get configmap myconfigmap
NAME          DATA      AGE
myconfigmap   1          12s
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl describe configmap myconfigmap
Name:         myconfigmap
Namespace:    default
Labels:       <none>
Annotations:  kubectl.kubernetes.io/last-applied-configuration={"apiVersion":"v1",
e-%timestamp...

Data
====
logging.conf:
-----
class: logging.handlers.RotatingFileHandler
formatter: precise
level: INFO
filename: %hostname-%timestamp.log

Events: <none>
ubuntu@k8s-master:~$
```

图10-21

在Pod中使用此ConfigMap，配置文件如图10-22所示。

```

apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: mypod
    image: busybox
    args:
      - /bin/sh
      - -c
      - sleep 10; touch /tmp/healthy; sleep 30000
    volumeMounts:
      - name: foo          ②
        mountPath: "/etc"
  volumes:
  - name: foo
    configMap:
      name: myconfigmap
      items:
        - key: logging.conf ①
          path: myapp/logging.conf

```

图10-22

① 在 volume 中指定存放配置信息的文件相对路径为 myapp/logging.conf。

② 将volume mount到容器的/etc目录。

创建Pod并读取配置信息，如图10-23所示。

```

ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl apply -f mypod.yml
pod "mypod" created
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl exec -it mypod sh
/ #
/ # cat /etc/myapp/logging.conf
class: logging.handlers.RotatingFileHandler
formatter: precise
level: INFO
filename: %hostname-%timestamp.log
/ #

```

图10-23

配置信息已经保存到/etc/myapp/logging.conf文件中。与Secret一样，Volume形式的ConfigMap也支持动态更新，留给大家自己实践。

10.5 小结

本章我们学习了如何向Pod传递配置信息。如果信息需要加密，可使用Secret；如果是一般的配置信息，则可使用ConfigMap。

Secret和ConfigMap支持四种定义方法。Pod在使用它们时，可以选择Volume方式或环境变量方式，不过只有Volume方式支持动态更新。

第11章 Helm—Kubernetes的包管理器

本章我们将学习Helm——Kubernetes的包管理器。

每个成功的软件平台都有一个优秀的打包系统，比如Debian、Ubuntu的apt，Red Hat、CentOS的yum。Helm则是Kubernetes上的包管理器。

本章我们将讨论为什么需要Helm、它的架构和组件，以及如何使用Helm。

11.1 Why Helm

Helm到底解决了什么问题？为什么Kubernetes需要Helm？

答案是：Kubernetes能够很好地组织和编排容器，但它缺少一个更高层次的应用打包工具，而Helm就是来干这件事的。

先来看个例子。

比如对于一个MySQL服务，Kubernetes需要部署下面这些对象：

(1) Service, 让外界能够访问到MySQL, 如图11-1所示。

```
apiVersion: v1
kind: Service
metadata:
  name: my-mysql
  labels:
    app: my-mysql
spec:
  ports:
    - name: mysql
      port: 3306
      targetPort: mysql
  selector:
    app: my-mysql
```

图11-1

(2) Secret, 定义MySQL的密码, 如图11-2所示。

```
apiVersion: v1
kind: Secret
metadata:
  name: my-mysql
  labels:
    app: my-mysql
type: Opaque
data:
  mysql-root-password: "M0MzREhRQWRjeQ=="
  mysql-password: "eGNXZkpMNmlkSw=="
```

图11-2

(3) PersistentVolumeClaim, 为MySQL申请持久化存储空间, 如图11-3所示。

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: my-mysql
  labels:
    app: my-mysql
spec:
  accessModes:
    - "ReadWriteOnce"
  resources:
    requests:
      storage: "8Gi"
```

图11-3

(4) Deployment, 部署MySQL Pod, 并使用上面的这些支持对象, 如图11-4所示。

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: my-mysql
  labels:
    app: my-mysql
spec:
  template:
    metadata:
      labels:
        app: my-mysql
    spec:
      containers:
        - name: my-mysql
          image: "mysql:5.7.14"
          env:
            - name: MYSQL_ROOT_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: my-mysql
                  key: mysql-root-password
            - name: MYSQL_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: my-mysql
                  key: mysql-password
            - name: MYSQL_USER
              value: ""
            - name: MYSQL_DATABASE
              value: ""
          ports:
            - name: mysql
              containerPort: 3306
          volumeMounts:
            - name: data
              mountPath: /var/lib/mysql
      volumes:
        - name: data
          persistentVolumeClaim:
            claimName: my-mysql
```

图11-4

我们可以将上面这些配置保存到对象各自的文件中，或者集中写进一个配置文件，然后通过`kubectl apply -f`部署。

到目前为止，**Kubernetes**对服务的部署支持得都挺好，如果应用只由一个或几个这样的服务组成，上面的部署方式完全足够了。

但是，如果我们开发的是微服务架构的应用，组成应用的服务可能多达十个甚至几十上百个，这种组织和管理应用的方式就不好使了：

（1）很难管理、编辑和维护如此多的服务。每个服务都有若干配置，缺乏一个更高层次的工具将这些配置组织起来。

（2）不容易将这些服务作为一个整体统一发布。部署人员需要首先理解应用都包含哪些服务，然后按照逻辑顺序依次执行`kubectl apply`，即缺少一种工具来定义应用与服务，以及服务与服务之间的依赖关系。

（3）不能高效地共享和重用服务。比如两个应用都要用到**MySQL**服务，但配置的参数不一样，这两个应用只能分别复制一套标准的**MySQL**配置文件，修改后通过`kubectl apply`部署。也就是说，不支持参数化配置和多环境部署。

（4）不支持应用级别的版本管理。虽然可以通过`kubectl rollout undo`进行回滚，但这只能针对单个**Deployment**，不支持整个应用的回滚。

（5）不支持对部署的应用状态进行验证。比如是否能够通过预定义的账号访问**MySQL**。虽然**Kubernetes**有健康检查，但那是针对单个容器，我们需要应用（服务）级别的健康检查。

Helm能够解决上面这些问题，**Helm**帮助**Kubernetes**成为微服务架构应用理想的部署平台。

11.2 Helm架构

在实践之前，我们先来看看**Helm**的架构。

Helm有两个重要的概念：`chart`和`release`。

- **chart**是创建一个应用的信息集合，包括各种Kubernetes对象的配置模板、参数定义、依赖关系、文档说明等。**chart**是应用部署的自包含逻辑单元。可以将**chart**想象成apt、yum中的软件安装包。
- **release**是**chart**的运行实例，代表了一个正在运行的应用。当**chart**被安装到Kubernetes集群，就生成一个**release**。**chart**能够多次安装到同一个集群，每次安装都是一个**release**。

Helm是包管理工具，这里的包就是指的**chart**。Helm能够：

- 从零创建新**chart**。
- 与存储**chart**的仓库交互，拉取、保存和更新**chart**。
- 在Kubernetes集群中安装和卸载**release**。
- 更新、回滚和测试**release**。

Helm包含两个组件：Helm客户端和Tiller服务器，如图11-5所示。

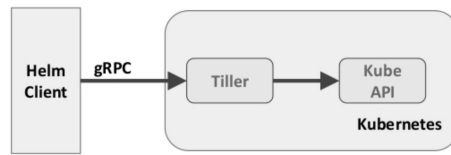


图11-5

Helm客户端是终端用户使用的命令行工具，用户可以：

- 在本地开发**chart**。
- 管理**chart**仓库。
- 与Tiller服务器交互。
- 在远程Kubernetes集群上安装**chart**。
- 查看**release**信息。
- 升级或卸载已有的**release**。

Tiller服务器运行在Kubernetes集群中，它会处理Helm客户端的请求，与Kubernetes API Server交互。Tiller服务器负责：

- 监听来自Helm客户端的请求。
- 通过**chart**构建**release**。
- 在Kubernetes中安装**chart**，并跟踪**release**的状态。
- 通过API Server升级或卸载已有的**release**。

简单地讲，Helm客户端负责管理chart，Tiller服务器负责管理release。

11.3 安装Helm

本节我们将依次安装Helm客户端和Tiller服务器。

11.3.1 Helm客户端

通常，我们将Helm客户端安装在能够执行kubectl命令的节点上，只需要下面一条命令：

```
curl https://raw.githubusercontent.com/kubernetes/helm/master/scripts/get | bash
```

结果如图11-6所示。

```
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ curl https://raw.githubusercontent.com/kubernetes/helm/master/scripts/get | bash  
% Total % Received % Xferd Average Speed Time Time Time Current  
Dload Upload Total Spent Left Speed  
100 6329 100 6329 0 0 4179 0 0:00:01 0:00:01 --:--:-- 4180  
Downloading https://kubernetes-helm.storage.googleapis.com/helm-v2.6.2-linux-amd64.tar.gz  
Preparing to install into /usr/local/bin  
helm installed into /usr/local/bin/helm  
Run 'helm init' to configure helm.  
ubuntu@k8s-master:~$
```

图11-6

执行helm version验证，如图11-7所示。

```
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ helm version  
Client: &version.Version{SemVer:"v2.7.0", GitCommit:"08c1144f5eb3e3b636  
d9775617287cc26e53dba4", GitTreeState:"clean"}  
Error: cannot connect to Tiller  
ubuntu@k8s-master:~$
```

图11-7

目前只能查看到客户端的版本，服务器还没有安装。

helm有很多子命令和参数，为了提高使用命令行的效率，通常建议安装helm的bash命令补全脚本，方法如下：

```
helm completion bash > .helmrc echo "source .helmrc" >> .bashrc
```

重新登录后就可以通过Tab键补全helm子命令和参数了，如图11-8所示。

```
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ helm  
completion  get      install  repo      status    version  
create      history  lint     reset     template  
delete      home     list     rollback  test  
dependency  init     package  search    upgrade  
fetch       inspect  plugin   serve     verify  
ubuntu@k8s-master:~$ helm install --  
--ca-file=      --name=          --tls-ca-cert=  
--cert-file=    --namespace=    --tls-cert=  
--debug         --name-template= --tls-key=  
--devel         --no-hooks      --tls-verify  
--dry-run       --replace       --values=  
--home=         --repo=         --verify  
--host=         --set=          --version=  
--key-file=     --tiller-namespace= --wait  
--keyring=      --timeout=  
--kube-context= --tls  
ubuntu@k8s-master:~$
```

图11-8

11.3.2 Tiller服务器

Tiller服务器安装非常简单，只需要执行helm init即可，如图11-9所示。

```
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ helm init  
$HELM_HOME has been configured at /home/ubuntu/.helm.  
  
Tiller (the Helm server-side component) has been installed into your Kubernetes Cluster.  
Happy Helming!  
ubuntu@k8s-master:~$
```

图11-9

Tiller本身也是作为容器化应用运行在Kubernetes Cluster中的，如图11-10所示。

```

ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get --namespace=kube-system svc tiller-deploy
NAME          CLUSTER-IP      EXTERNAL-IP      PORT(S)          AGE
tiller-deploy 10.98.124.71    <none>           44134/TCP        1m
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get --namespace=kube-system deployment tiller-deploy
NAME          DESIRED    CURRENT    UP-TO-DATE    AVAILABLE    AGE
tiller-deploy 1          1          1             1            1m
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get --namespace=kube-system pod tiller-deploy-1936853538-c8sfp
NAME          READY    STATUS    RESTARTS    AGE
tiller-deploy-1936853538-c8sfp 1/1      Running   0           2m
ubuntu@k8s-master:~$

```

图11-10

可以看到Tiller的Service、Deployment和Pod。

现在，helm version已经能够查看到服务器的版本信息了，如图11-11所示。

```

root@k8s-master:~# helm version
Client: &version.Version{SemVer:"v2.7.0", GitCommit:"08c1144f5eb3e3b636d9775617287cc26e53dba4", GitTreeState:"clean"}
Server: &version.Version{SemVer:"v2.7.0", GitCommit:"08c1144f5eb3e3b636d9775617287cc26e53dba4", GitTreeState:"clean"}

```

图11-11

11.4 使用Helm

Helm安装成功后，可执行helm search查看当前可安装的chart，如图11-12所示。

```

ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ helm search
NAME          VERSION    DESCRIPTION
local/cool-chart 0.1.0      A Helm chart for Kubernetes
stable/acs-engine-autoscaler 2.1.0      Scales worker nodes within agent pools
stable/artifactory 6.1.0      Universal Repository Manager supporting all maj...
stable/aws-cluster-autoscaler 0.3.1      Scales worker nodes within autoscaling groups.
stable/buildkite 0.2.0      Agent for Buildkite
stable/centrifugo 2.0.0      Centrifugo is a real-time messaging server.
stable/chaoskube 0.5.0      Chaoskube periodically kills random pods in you...
stable/chronograf 0.3.0      Open-source web application written in Go and R...
stable/cluster-autoscaler 0.2.0      Scales worker nodes within autoscaling groups.
stable/cockroachdb 0.5.0      CockroachDB is a scalable, survivable, strongly...
stable/concourse 0.7.0      Concourse is a simple and scalable CI system.
stable/consul 0.4.2      Highly available and distributed service discov...
stable/coredns 0.5.0      CoreDNS is a DNS server that chains middleware ...
stable/coscale 0.2.0      CoScale Agent
stable/dask-distributed 2.0.0      Distributed computation in Python
stable/datadog 0.8.0      DataDog Agent
stable/dokuwiki 0.2.0      DokuWiki is a standards-compliant, simple to us...
stable/drupal 0.10.2     One of the most versatile open source content m...
stable/etcd-operator 0.5.0      CoreOS etcd-operator Helm chart for Kubernetes

```

图11-12

这个列表很长，这里只截取了一部分。大家不禁会问，这些chart都是从哪里来的？

前面说过，Helm可以像apt和yum管理软件包一样管理chart。apt和yum的软件包存放在仓库中，同样的，Helm也有仓库，如图11-13所示。

```
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ helm repo list  
NAME      URL  
stable    https://kubernetes-charts.storage.googleapis.com  
local     http://127.0.0.1:8879/charts  
ubuntu@k8s-master:~$
```

图11-13

Helm安装时已经默认配置好了两个仓库：stable和local。stable是官方仓库，local是用户存放自己开发的chart的本地仓库。

helm search会显示chart位于哪个仓库，比如local/cool-chart和stable/acse-engineautoscaler。

用户可以通过helm repo add添加更多的仓库，比如企业的私有仓库，仓库的管理和维护方法请参考官网文档<https://docs.helm.sh>。

与apt和yum一样，helm也支持关键字搜索，如图11-14所示。包括DESCRIPTION在内的所有信息，只要跟关键字匹配，都会显示在结果列表中。

```
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ helm search mysql  
NAME                VERSION DESCRIPTION  
stable/mysql         0.3.0   Fast, reliable, scalable, and easy to use open-...  
stable/percona       0.3.0   free, fully compatible, enhanced, open source d...  
stable/gcloud-sqlproxy 0.2.0   Google Cloud SQL Proxy  
stable/mariadb       2.0.0   Fast, reliable, scalable, and easy to use open-...  
ubuntu@k8s-master:~$
```

图11-14

安装chart也很简单，执行如下命令就可以安装MySQL。

```
helm install stable/mysql
```

如果看到如图11-15所示的报错，通常是因为Tiller服务器的权限不足。

A terminal window with a dark background and light text. It shows a user at a prompt 'ubuntu@k8s-master:~\$' typing 'helm install stable/mysql'. The output is 'Error: no available release name found', followed by the prompt again.

```
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ helm install stable/mysql  
Error: no available release name found  
ubuntu@k8s-master:~$
```

图11-15

执行如下命名添加权限：

```
kubectl create serviceaccount --namespace kube-system tiller
```

```
kubectl create clusterrolebinding tiller-cluster-rule --  
clusterrole=cluster-admin --serviceaccount=kube-system:tiller
```

```
kubectl patch deploy --namespace kube-system tiller-deploy -  
p '{"spec":{"template":{"spec":{"serviceAccount":"tiller"}}}}'
```

然后再次执行下面的命令，结果如图11-16所示。

```
helm install stable/mysql
```

```

ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ helm install stable/mysql
NAME:      fun-zorse ①
LAST DEPLOYED: Tue Oct 17 11:34:55 2017
NAMESPACE: default
STATUS: DEPLOYED

RESOURCES: ②
==> v1/Service
NAME                CLUSTER-IP   EXTERNAL-IP   PORT(S)    AGE
fun-zorse-mysql     10.108.23.5  <none>        3306/TCP   0s

==> v1beta1/Deployment
NAME                DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
fun-zorse-mysql     1         1         1             0           0s

==> v1/Secret
NAME                TYPE      DATA   AGE
fun-zorse-mysql     Opaque    2       1s

==> v1/PersistentVolumeClaim
NAME                STATUS     VOLUME   CAPACITY   ACCESSMODES   STORAGECLASS   AGE
fun-zorse-mysql     Pending   1s

NOTES: ③
MySQL can be accessed via port 3306 on the following DNS name from within your cluster:
fun-zorse-mysql.default.svc.cluster.local

To get your root password run:

    kubectl get secret --namespace default fun-zorse-mysql -o jsonpath="{.data.mysql-root-password}"

To connect to your database:

1. Run an Ubuntu pod that you can use as a client:

    kubectl run -i --tty ubuntu --image=ubuntu:16.04 --restart=Never -- bash -il

2. Install the mysql client:

    $ apt-get update && apt-get install mysql-client -y

3. Connect using the mysql cli, then provide your password:

    $ mysql -h fun-zorse-mysql -p

```

图11-16

输出分为三部分：

① chart本次部署的描述信息。

- NAME是release的名字，因为我们没用-n参数指定，所以Helm随机生成了一个，这里是fun-zorse。
- NAMESPACE是release部署的namespace，默认是default，也可以通过--namespace指定。
- STATUS为DEPLOYED，表示已经将chart部署到集群。

② 当前 release 包含的资源：Service、Deployment、Secret 和 PersistentVolumeClaim，其名字都是fun-zorse-mysql，命名的格式为ReleaseName-ChartName。

③ NOTES部分显示的是release的使用方法，比如如何访问Service、如何获取数据库密码以及如何连接数据库等。

通过kubectl get可以查看组成release的各个对象，如图11-17所示。

```
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ kubectl get service fun-zorse-mysql  
NAME             CLUSTER-IP   EXTERNAL-IP   PORT(S)    AGE  
fun-zorse-mysql  10.108.23.5  <none>        3306/TCP   15m  
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ kubectl get deployment fun-zorse-mysql  
NAME             DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE  
fun-zorse-mysql  1         1         1            0           15m  
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ kubectl get pod fun-zorse-mysql-2959529493-hfq1t  
NAME             READY     STATUS    RESTARTS   AGE  
fun-zorse-mysql-2959529493-hfq1t  0/1      Pending   0          15m  
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ kubectl get pvc fun-zorse-mysql  
NAME             STATUS    VOLUME   CAPACITY   ACCESSMODES   STORAGECLASS   AGE  
fun-zorse-mysql  Pending                                     default        15m  
ubuntu@k8s-master:~$
```

图11-17

因为我们还没有准备PersistentVolume，所以当前release还不可用。

helm list显示已经部署的release，helm delete可以删除release，如图11-18所示。

```
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ helm list  
NAME      REVISION   UPDATED                     STATUS   CHART       NAMESPACE  
fun-zorse 1          Tue Oct 17 11:34:55 2017 DEPLOYED mysql-0.3.0 default  
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ helm delete fun-zorse  
release "fun-zorse" deleted  
ubuntu@k8s-master:~$
```

图11-18

Helm的使用方法像极了apt和yum，用Helm来管理Kubernetes应用非常方便。

11.5 chart详解

chart是Helm的应用打包格式。chart由一系列文件组成，这些文件描述了Kubernetes部署应用时所需要的资源，比如Service、Deployment、PersistentVolumeClaim、Secret、ConfigMap等。

单个的chart可以非常简单，只用于部署一个服务，比如Memcached。chart也可以很复杂，部署整个应用，比如包含HTTP Servers、Database、消息中间件、Cache等。

chart将这些文件放置在预定义的目录结构中，通常整个chart被打成tar包，而且标注上版本信息，便于Helm部署。

下面我们将详细讨论chart的目录结构以及包含的各类文件。

11.5.1 chart目录结构

以前面MySQL chart为例。一旦安装了某个chart，我们就可以在`~/.helm/cache/archive`中找到chart的tar包，如图11-19所示。

```
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ ls ~/.helm/cache/archive  
mysql-0.3.0.tgz  
ubuntu@k8s-master:~$
```

图11-19

解压后，MySQL chart目录结构如图11-20所示。

```
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ tree mysql  
mysql  
├── Chart.yaml  
├── README.md  
├── templates  
│   ├── configmap.yaml  
│   ├── deployment.yaml  
│   ├── _helpers.tpl  
│   ├── NOTES.txt  
│   ├── pvc.yaml  
│   ├── secrets.yaml  
│   └── svc.yaml  
└── values.yaml  
  
1 directory, 10 files  
ubuntu@k8s-master:~$
```

图11-20

目录名就是chart的名字（不带版本信息），这里是mysql，包含如下内容。

(1) Chart.yaml

YAML文件，描述chart的概要信息，如图11-21所示。

```
description: Fast, reliable, scalable, and easy to use open-source relational database
  system.
engine: gotpl
home: https://www.mysql.com/
icon: https://www.mysql.com/common/logos/logo-mysql-170x115.png
keywords:
- mysql
- database
- sql
maintainers:
- email: viglesias@google.com
  name: Vic Iglesias
name: mysql
sources:
- https://github.com/kubernetes/charts
- https://github.com/docker-library/mysql
version: 0.3.0
```

图11-21

name和version是必填项，其他都是可选项。

(2) README.md

Markdown格式的README文件，相当于chart的使用文档，此文件为可选，如图11-22所示。

```

# MySQL

[MySQL](https://MySQL.org) is one of the most popular database servers in the world. Notable users includ

## Introduction

This chart bootstraps a single node MySQL deployment on a [Kubernetes](http://kubernetes.io) cluster usin

## Prerequisites

- Kubernetes 1.6+ with Beta APIs enabled
- PV provisioner support in the underlying infrastructure

## Installing the Chart

To install the chart with the release name `my-release`:

```bash
$ helm install --name my-release stable/mysql
```

The command deploys MySQL on the Kubernetes cluster in the default configuration. The [configuration](#co

By default a random password will be generated for the root user. If you'd like to set your own password
in the values.yaml.

You can retrieve your root password by running the following command. Make sure to replace [YOUR_RELEASE]

    printf $(printf '\%o' `kubectl get secret [YOUR_RELEASE_NAME]-mysql -o jsonpath="{.data.mysql-root-pa

> Tip: List all releases using `helm list`

## Uninstalling the Chart

To uninstall/delete the `my-release` deployment:

```bash
$ helm delete my-release
```

The command removes all the Kubernetes components associated with the chart and deletes the release.

## Configuration

The following tables lists the configurable parameters of the MySQL chart and their default values.



Parameter	Description	Default
<code>imageTag</code>	<code>mysql</code> image tag.	Most recent release
<code>imagePullPolicy</code>	Image pull policy	<code>IfNotPresent</code>
<code>mysqlRootPassword</code>	Password for the <code>root</code> user.	<code>nil</code>
<code>mysqlUser</code>	Username of new user to create.	<code>nil</code>
<code>mysqlPassword</code>	Password for the new user.	<code>nil</code>
<code>mysqlDatabase</code>	Name for new database to create.	<code>nil</code>


```

图11-22

(3) LICENSE

文本文件，描述chart的许可信息，此文件为可选。

(4) requirements.yaml

chart可能依赖其他的chart，这些依赖关系可通过requirements.yaml指定，如图11-23所示。

```
dependencies:
- name: rabbitmq
  version: 1.2.3
  repository: http://example.com/charts
- name: memcached
  version: 3.2.1
  repository: http://another.example.com/charts
```

图11-23

在安装过程中，依赖的chart也会被一起安装。

(5) values.yaml

chart支持在安装时根据参数进行定制化配置，而values.yaml则提供了这些配置参数的默认值，如图11-24所示。

```
## mysql image version
image: "mysql"
imageTag: "5.7.14"

## Specify password for root user
##
## Default: random 10 character string
# mysqlRootPassword: testing

## Create a database user
##
# mysqlUser:
# mysqlPassword:

imagePullPolicy: IfNotPresent

## Persist data to a persistent volume
persistence:
  enabled: true
  ## If defined, storageClassName: <storageClass>
  ## If set to "-", storageClassName: "", which disables dynamic provisioning
  ##
  # storageClass: "-"
  accessMode: ReadWriteOnce
  size: 8Gi

## Configure resource requests and limits
##
resources:
  requests:
    memory: 256Mi
    cpu: 100m

# Custom mysql configuration files used to override default mysql settings
configurationFiles:
# mysql.cnf: |-
#   [mysqld]
#   skip-name-resolve
```

图11-24

(6) templates目录

各类Kubernetes资源的配置模板都放置在这里。Helm会将values.yaml中的参数值注入模板中，生成标准的YAML配置文件。

模板是chart最重要的部分，也是Helm最强大的地方。模板增加了应用部署的灵活性，能够适用不同的环境，我们后面会详细讨论。

(7) templates/NOTES.txt

chart的简易使用文档，chart安装成功后会显示此文档内容，如图11-25所示。

```
MySQL can be accessed via port 3306 on the following DNS name from within your cluster:
{{ template "mysql.fullname" . }}.{{ .Release.Namespace }}.svc.cluster.local

To get your root password run:

    kubectl get secret --namespace {{ .Release.Namespace }} {{ template "mysql.fullname" . }}

To connect to your database:

1. Run an Ubuntu pod that you can use as a client:

    kubectl run -i --tty ubuntu --image=ubuntu:16.04 --restart=Never -- bash -il

2. Install the mysql client:

    $ apt-get update && apt-get install mysql-client -y

3. Connect using the mysql cli, then provide your password:
    $ mysql -h {{ template "mysql.fullname" . }} -p
```

图11-25

与模板一样，可以在NOTE.txt中插入配置参数，Helm会动态注入参数值。

11.5.2 chart模板

Helm通过模板创建Kubernetes能够理解的YAML格式的资源配置文件，我们将通过例子来学习如何使用模板。

以templates/secrets.yaml为例，如图11-26所示。

```

apiVersion: v1
kind: Secret
metadata:
  name: {{ template "mysql.fullname" . }} ①
  labels:
    app: {{ template "mysql.fullname" . }}
    chart: "{{ .Chart.Name }}" - {{ .Chart.Version }}
    release: "{{ .Release.Name }}" ②
    heritage: "{{ .Release.Service }}"
type: Opaque
data:
  {{ if .Values.mysqlRootPassword }} ③
  mysql-root-password: {{ .Values.mysqlRootPassword | b64enc | quote }}
  {{ else }}
  mysql-root-password: {{ randAlphaNum 10 | b64enc | quote }}
  {{ end }}
  {{ if .Values.mysqlPassword }}
  mysql-password: {{ .Values.mysqlPassword | b64enc | quote }}
  {{ else }}
  mysql-password: {{ randAlphaNum 10 | b64enc | quote }}
  {{ end }}

```

图11-26

从结构上看，文件的内容非常像Secret配置，只是大部分属性值变成了`{{ xxx }}`。这些`{{ xxx }}`实际上是模板的语法。Helm采用了Go语言的模板来编写chart。Go模板非常强大，支持变量、对象、函数、流控制等功能。下面我们通过解析`templates/secrets.yaml`快速学习模板。

① `{{ template "mysql.fullname" . }}`定义Secret的name。关键字`template`的作用是引用一个子模板`mysql.fullname`。这个子模板是在`templates/_helpers.tpl`文件中定义的，如图11-27所示。

```

{{/* vim: set filetype=mustache: */}}
{{/*
Expand the name of the chart.
*/}}
{{- define "mysql.name" -}}
{{- default .Chart.Name .Values.nameOverride | trunc 63 | trimSuffix "-" -}}
{{- end -}}

{{/*
Create a default fully qualified app name.
We truncate at 63 chars because some Kubernetes name fields are limited to this
*/}}
{{- define "mysql.fullname" -}}
{{- $name := default .Chart.Name .Values.nameOverride -}}
{{- printf "%s-%s" .Release.Name $name | trunc 63 | trimSuffix "-" -}}
{{- end -}}

```

图11-27

这个定义还是很复杂的，因为它用到了模板语言中的对象、函数、流控制等概念。现在看不懂没关系，这里我们学习的重点是：如果存在一些信息多个模板都会用到，则可在`templates/_helpers.tpl`中将其定义为子模板，然后通过`templates`函数引用。

这里`mysql.fullname`是由`release`与`chart`二者名字拼接组成的。

根据`chart`的最佳实践，所有资源的名称都应该保持一致。对于我们这个`chart`，无论`Secret`还是`Deployment`、`PersistentVolumeClaim`、`Service`，它们的名称都是子模板`mysql.fullname`的值。

② `Chart`和`Release`是Helm预定义的对象，每个对象都有自己的属性，可以在模板中使用。如果使用下面的命令安装`chart`：

```
helm install stable/mysql -n my
```

那么：

- `{{ .Chart.Name }}`的值为`mysql`。
- `{{ .Chart.Version }}`的值为`0.3.0`。
- `{{ .Release.Name }}`的值为`my`。
- `{{ .Release.Service }}`始终取值为`Tiller`。
- `{{ template "mysql.fullname" . }}`计算结果为`my-mysql`。

③ 这里指定`mysql-root-password`的值，不过使用了`if-else`的流控制，其逻辑为：如果`.Values.mysqlRootPassword`有值，就对其进行`base64`编码，否则随机生成一个10位的字符串并编码。

`Values`也是预定义的对象，代表的是`values.yaml`文件。`.Values.mysqlRootPassword`则是`values.yaml`中定义的`mysqlRootPassword`参数，如图11-28所示。

```
## mysql image version
##
image: "mysql"
imageTag: "5.7.14"

## Specify password for root user
##
## Default: random 10 character string
# mysqlRootPassword: testing

## Create a database user
##
# mysqlUser:
# mysqlPassword:
```

图11-28

因为mysqlRootPassword被注释掉了，没有赋值，所以逻辑判断会走else，即随机生成密码。

randAlphaNum、b64enc、quote都是Go模板语言支持的函数，函数之间可以通过管道|连接。{{ randAlphaNum 10 | b64enc | quote }}的作用是首先随机产生一个长度为10的字符串，然后将其base64编码，最后两边加上双引号。

templates/secrets.yaml这个例子展示了chart模板主要的功能，我们最大的收获应该是：模板将chart参数化了，通过values.yaml可以灵活定制应用。

无论多复杂的应用，用户都可以用Go模板语言编写出chart。无非是使用到更多的函数、对象和流控制。对于初学者，建议尽量参考官方的chart。根据二八定律，这些chart已经覆盖了绝大部分情况，而且采用了最佳实践。如何遇到不懂的函数、对象和其他语法，可参考官网文档<https://docs.helm.sh>。

11.5.3 再次实践MySQL chart

学习了chart结构和模板的知识后，现在重新实践一次MySQL chart，相信会有更多收获。

1. chart安装前的准备

作为准备工作，安装之前需要先清楚chart的使用方法。这些信息通常记录在values.yaml和README.md中。除了下载源文件查看，执行helm inspect values可能是更方便的方法，如图11-29所示。

```
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ helm inspect values stable/mysql  
## mysql image version  
## ref: https://hub.docker.com/r/library/mysql/tags/  
##  
image: "mysql"  
imageTag: "5.7.14"  
  
## Specify password for root user  
##  
## Default: random 10 character string  
# mysqlRootPassword: testing  
  
## Create a database user  
##  
# mysqlUser:  
# mysqlPassword:  
  
## Allow unauthenticated access, uncomment to enable  
##  
# mysqlAllowEmptyPassword: true
```

图11-29

输出的实际上是values.yaml的内容。阅读注释就可以知道MySQL chart支持哪些参数，安装之前需要做哪些准备。其中有一部分是关于存储的，如图11-30所示。

```
persistence:  
  enabled: true  
  ## database data Persistent Volume Storage Class  
  ## If defined, storageClassName: <storageClass>  
  ## If set to "-", storageClassName: "", which disables dynamic provisioning  
  ## If undefined (the default) or set to null, no storageClassName spec is  
  ## set, choosing the default provisioner. (gp2 on AWS, standard on  
  ## GKE, AWS & OpenStack)  
  ##  
  # storageClass: "-"  
  accessMode: ReadWriteOnce  
  size: 8Gi
```

图11-30

chart定义了一个PersistentVolumeClaim，申请8GB的PersistentVolume。由于我们的实验环境不支持动态供给，因此要预先创建好相应的PV，其配置文件mysql-pv.yml的内容如图11-31所示。

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: mysql-pv
spec:
  accessModes:
    - ReadWriteOnce
  capacity:
    storage: 8Gi
  persistentVolumeReclaimPolicy: Retain
# storageClassName: nfs
nfs:
  path: /nfsdata/mysql-pv
  server: 192.168.56.105
```

图11-31

创建PV mysql-pv，如图11-32所示。

```
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl apply -f mysql-pv.yml
persistentvolume "mysql-pv" created
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get pv
NAME          CAPACITY  ACCESSMODES  RECLAIMPOLICY  STATUS    CLAIM    STORAGECLASS
mysql-pv      8Gi       RWO          Retain         Available
ubuntu@k8s-master:~$
```

图11-32

接下来就可以安装chart了。

2. 定制化安装chart

除了接受values.yaml的默认值，我们还可以定制化chart，比如设置mysqlRootPassword。

Helm有两种方式传递配置参数：

(1) 指定自己的values文件。通常的做法是首先通过helm inspect values mysql >myvalues.yaml 生成 values 文件，然后设置mysqlRootPassword，最后执行helm install--values=myvalues.yaml mysql。

(2) 通过--set直接传入参数值，如图11-33所示。

```
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ helm install stable/mysql --set mysqlRootPassword=abc123 -n my  
NAME: my  
LAST DEPLOYED: Thu Oct 19 14:24:56 2017  
NAMESPACE: default  
STATUS: DEPLOYED  
  
RESOURCES:  
==> v1/Service  
NAME          CLUSTER-IP    EXTERNAL-IP  PORT(S)    AGE  
my-mysql      10.104.59.139 <none>       3306/TCP   0s  
  
==> v1beta1/Deployment  
NAME          DESIRED  CURRENT  UP-TO-DATE  AVAILABLE  AGE  
my-mysql      1        1        1            0           0s  
  
==> v1/Secret  
NAME          TYPE      DATA  AGE  
my-mysql      Opaque    2      0s  
  
==> v1/PersistentVolumeClaim  
NAME          STATUS  VOLUME  CAPACITY  ACCESSMODES  STORAGECLASS  AGE  
my-mysql      Pending 0s
```

图11-33

mysqlRootPassword设置为abc123。另外，-n设置release为my，各类资源的名称即为my-mysql。

通过helm list和helm status可以查看chart的最新状态，如图11-34所示。

```
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ helm list  
NAME      REVISION      UPDATED              STATUS      CHART          NAMESPACE  
my        1             Thu Oct 19 14:24:56 2017    DEPLOYED   mysql-0.3.0    default  
  
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ helm status my  
LAST DEPLOYED: Thu Oct 19 14:24:56 2017  
NAMESPACE: default  
STATUS: DEPLOYED  
  
RESOURCES:  
==> v1/Secret  
NAME          TYPE      DATA  AGE  
my-mysql      Opaque    2      8m  
  
==> v1/PersistentVolumeClaim  
NAME          STATUS    VOLUME  CAPACITY  ACCESSMODES  STORAGECLASS  AGE  
my-mysql      Bound    mysql-pv  8Gi       RWO           mysql-pv      8m  
  
==> v1/Service  
NAME          CLUSTER-IP    EXTERNAL-IP  PORT(S)    AGE  
my-mysql      10.104.59.139 <none>       3306/TCP   8m  
  
==> v1beta1/Deployment  
NAME          DESIRED  CURRENT  UP-TO-DATE  AVAILABLE  AGE  
my-mysql      1        1        1            1           8m
```

图11-34

PVC已经Bound，Deployment也变为AVAILABLE。

11.5.4 升级和回滚release

release发布后可以执行helm upgrade对其进行升级，通过--values或--set应用新的配置。比如将当前的MySQL版本升级到5.7.15，如图11-35所示。

```
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ helm upgrade --set imageTag=5.7.15 my stable/mysql  
Release "my" has been upgraded. Happy Helming!  
LAST DEPLOYED: Thu Oct 19 14:37:46 2017  
NAMESPACE: default  
STATUS: DEPLOYED  
  
RESOURCES:  
==> v1/Secret  
NAME      TYPE      DATA  AGE  
my-mysql  Opaque    2      12m  
  
==> v1/PersistentVolumeClaim  
NAME      STATUS    VOLUME   CAPACITY   ACCESSMODES  STORAGECLASS  AGE  
my-mysql  Bound     mysql-pv  8Gi        RWO           mysql-pv      12m  
  
==> v1/Service  
NAME      CLUSTER-IP    EXTERNAL-IP  PORT(S)    AGE  
my-mysql  10.104.59.139 <none>        3306/TCP   12m  
  
==> v1beta1/Deployment  
NAME      DESIRED  CURRENT  UP-TO-DATE  AVAILABLE  AGE  
my-mysql  1        1        1            0          12m
```

图11-35

等待一些时间，升级成功，如图11-36所示。

```
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ kubectl get deployment my-mysql -o wide  
NAME      DESIRED  CURRENT  UP-TO-DATE  AVAILABLE  AGE    CONTAINER(S)  IMAGE(S)  SELECTED  
my-mysql  1        1        1            1          5m     my-mysql      mysql:5.7.15  app=
```

图11-36

helm history可以查看release所有的版本。通过helm rollback可以回滚到任何版本，如图11-37所示。

```

ubuntu@k8s-master:~$ helm history my
REVISION      UPDATED              STATUS      CHART          DESCRIPTION
1             Thu Oct 19 11:18:55 2017    SUPERSEDED  mysql-0.3.0    Install complete
2             Thu Oct 19 11:20:01 2017    DEPLOYED    mysql-0.3.0    Upgrade complete
ubuntu@k8s-master:~$ helm rollback my 1
Rollback was a success! Happy Helming!
ubuntu@k8s-master:~$

```

图11-37

回滚成功，MySQL恢复到5.7.14，如图11-38所示。

```

ubuntu@k8s-master:~$ helm history my
REVISION      UPDATED              STATUS      CHART          DESCRIPTION
1             Thu Oct 19 11:18:55 2017    SUPERSEDED  mysql-0.3.0    Install complete
2             Thu Oct 19 11:20:01 2017    SUPERSEDED  mysql-0.3.0    Upgrade complete
3             Thu Oct 19 11:27:01 2017    DEPLOYED    mysql-0.3.0    Rollback to 1
ubuntu@k8s-master:~$ kubectl get deployment my-mysql -o wide
NAME          DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE   CONTAINER(S)   IMAGE(S)           SELECTED
my-mysql      1         1         1            1           8m    my-mysql       mysql:5.7.14      app
ubuntu@k8s-master:~$

```

图11-38

11.5.5 开发自己的chart

Kubernetes给我们提供了大量官方chart，不过要部署微服务应用，还是需要开发自己的chart，下面就来实践这个主题。

1. 创建chart

执行helm create mychart命令，创建chart mychart，如图11-39所示。

```
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ helm create mychart  
Creating mychart  
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ tree mychart  
mychart  
├── charts  
├── Chart.yaml  
├── templates  
│   ├── deployment.yaml  
│   ├── _helpers.tpl  
│   ├── ingress.yaml  
│   ├── NOTES.txt  
│   └── service.yaml  
└── values.yaml  
  
2 directories, 7 files  
ubuntu@k8s-master:~$
```

图11-39

Helm会帮我们创建目录mychart，并生成各类chart文件。这样我们就可以在此基础上开发自己的chart了。

新建的chart默认包含一个nginx应用示例，values.yaml内容如图11-40所示。

```
# Default values for mychart.  
# This is a YAML-formatted file.  
# Declare variables to be passed into your templates.  
replicaCount 1  
image:  
  repository: nginx  
  tag: stable  
  pullPolicy: IfNotPresent  
service:  
  name: nginx  
  type: ClusterIP  
  externalPort: 80  
  internalPort: 80  
ingress:  
  enabled: false  
  # Used to create an Ingress record.  
  hosts:  
    - chart-example.local
```

图11-40

开发时建议大家参考官方chart中的模板values.yaml、Chart.yaml，里面包含了大量最佳实践和最常用的函数、流控制，这里就不一一展开了。

2. 调试chart

只要是程序就会有bug，chart也不例外。Helm提供了debug的工具：`helm lint`和`helm install --dry-run --debug`。

`helm lint`会检测chart的语法，报告错误以及给出建议。

比如我们故意在values.yaml的第8行漏掉了行尾的那个冒号，如图11-41所示。

```
1 # Default values for mychart.
2 # This is a YAML-formatted file.
3 # Declare variables to be passed into your templates.
4 replicaCount: 1
5 image:
6   repository: nginx
7   tag: stable
8   pullPolicy IfNotPresent
9 service:
10  name: nginx
11  type: ClusterIP
12  externalPort: 80
13  internalPort: 80
```

图11-41

`helm lint mychart`会指出这个语法错误，如图11-42所示。

```
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ helm lint mychart
==> Linting mychart
[INFO] Chart.yaml: icon is recommended
[ERROR] values.yaml: unable to parse YAML
       error converting YAML to JSON: yaml: line 8: could not find expected ':'
Error: 1 chart(s) linted, 1 chart(s) failed
ubuntu@k8s-master:~$
```

图11-42

mychart目录被作为参数传递给helm lint。错误修复后则能通过检测，如图11-43所示。

```
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ helm lint mychart  
==> Linting mychart  
[INFO] Chart.yaml: icon is recommended  
  
1 chart(s) linted, no failures  
ubuntu@k8s-master:~$
```

图11-43

helm install --dry-run --debug会模拟安装chart，并输出每个模板生成的YAML内容，如图11-44、图11-45所示。

```
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ helm install --dry-run mychart --debug  
[debug] Created tunnel using local port: '46295'  
  
[debug] SERVER: "localhost:46295"  
  
[debug] Original chart version: ""  
[debug] CHART PATH: /home/ubuntu/mychart  
  
NAME:    solitary-kudu  
REVISION: 1  
RELEASED: Fri Oct 20 09:59:51 2017  
CHART: mychart-0.1.0  
USER-SUPPLIED VALUES:  
{}  
  
COMPUTED VALUES:  
image:  
  pullPolicy: IfNotPresent  
  repository: nginx  
  tag: stable  
ingress:  
  annotations: null  
  enabled: false  
  hosts:  
  - chart-example.local  
  tls: null  
replicaCount: 1  
resources: {}  
service:  
  externalPort: 80  
  internalPort: 80  
  name: nginx  
  type: ClusterIP  
  
HOOKS:  
MANIFEST:
```

图11-44

```

---
# Source: mychart/templates/service.yaml
apiVersion: v1
kind: Service
metadata:
  name: solitary-kudu-mychart
  labels:
    app: mychart
    chart: mychart-0.1.0
    release: solitary-kudu
    heritage: Tiller
spec:
  type: ClusterIP
  ports:
    - port: 80
      targetPort: 80
      protocol: TCP
      name: nginx
  selector:
    app: mychart
    release: solitary-kudu
---
# Source: mychart/templates/deployment.yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: solitary-kudu-mychart
  labels:
    app: mychart
    chart: mychart-0.1.0
    release: solitary-kudu
    heritage: Tiller
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: mychart
        release: solitary-kudu
    spec:
      containers:
        - name: mychart
          image: "nginx:stable"
          imagePullPolicy: IfNotPresent
          ports:
            - containerPort: 80
          livenessProbe:
            httpGet:
              path: /
              port: 80
          readinessProbe:
            httpGet:
              path: /
              port: 80
          resources:
            {}

```

图11-45

我们可以检测这些输出，判断是否与预期相符。

同样，mychart目录作为参数传递给helm install --dry-run --debug。

3. 安装chart

当我们准备就绪，就可以安装chart了。Helm支持四种安装方法：

- (1) 安装仓库中的chart，例如helm install stable/nginx。
- (2) 通过tar包安装，例如helm install ./nginx-1.2.3.tgz。
- (3) 通过chart本地目录安装，例如helm install ./nginx。
- (4) 通过URL安装，例如helm install https://example.com/charts/nginx-1.2.3.tgz。

这里我们使用本地目录安装，如图11-46所示。

```
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ helm install mychart  
NAME:      opinionated-nightingale  
LAST DEPLOYED: Fri Oct 20 10:14:06 2017  
NAMESPACE: default  
STATUS: DEPLOYED  
  
RESOURCES:  
==> v1/Service  
NAME                                CLUSTER-IP      EXTERNAL-IP  PORT(S)  AGE  
opinionated-nightingale-mychart    10.100.28.221   <none>       80/TCP   0s  
  
==> v1beta1/Deployment  
NAME                                DESIRED  CURRENT  UP-TO-DATE  AVAILABLE  AGE  
opinionated-nightingale-mychart    1        1        1            0          0s  
  
NOTES:  
1. Get the application URL by running these commands:  
   export POD_NAME=$(kubectl get pods --namespace default -l "app=mychart,release=opi  
   echo "Visit http://127.0.0.1:8080 to use your application"  
   kubectl port-forward $POD_NAME 8080:80  
ubuntu@k8s-master:~$
```

图11-46

当 chart 部署到 Kubernetes 集群后，便可以对其进行更为全面的测试了。

4. 将chart添加到仓库

chart通过测试后可以添加到仓库中，团队其他成员就能够使用了。任何 HTTP Server 都可以用作 chart 仓库。下面演示在 k8s-node1 192.168.56.106上搭建仓库。

- (1) 在k8s-node1上启动一个httpd容器，如图11-47所示。

```
root@k8s-node1:~#  
root@k8s-node1:~# mkdir /var/www  
root@k8s-node1:~#  
root@k8s-node1:~# docker run -d -p 8080:80 -v /var/www:/usr/local/apache2/htdocs/ httpd  
f3d5f0779a04a3074bd332764263e0283d8548e8f9b2b96dc744e098b45ce075  
root@k8s-node1:~#
```

图11-47

- (2) 通过helm package将mychart打包，如图11-48所示。

```
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ helm package mychart  
Successfully packaged chart and saved it to: /home/ubuntu/mychart-0.1.0.tgz  
ubuntu@k8s-master:~$
```

图11-48

- (3) 执行helm repo index生成仓库的index文件，如图11-49所示。

```
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ mkdir myrepo  
ubuntu@k8s-master:~$ mv mychart-0.1.0.tgz myrepo/  
ubuntu@k8s-master:~$ helm repo index myrepo/ --url http://192.168.56.106:8080/charts  
ubuntu@k8s-master:~$ ls myrepo/  
index.yaml mychart-0.1.0.tgz  
ubuntu@k8s-master:~$
```

图11-49

Helm会扫描myrepo目录中的所有tgz包并生成index.yaml。--url指定的是新仓库的访问路径。新生成的index.yaml记录了当前仓库中所有chart的信息，如图11-50所示。

```

apiVersion: v1
entries:
  mychart:
    - apiVersion: v1
      created: 2017-10-21T16:39:39.184818935+08:00
      description: A Helm chart for Kubernetes
      digest: ae8d7138002d432014dc8638ec37202823e9207445caf08a660d154b26e936ea
      name: mychart
      urls:
        - http://192.168.56.106:8080/charts/mychart-0.1.0.tgz
      version: 0.1.0
generated: 2017-10-21T16:39:39.184265707+08:00

```

图11-50

当前只有mychart这一个chart。

(4) 将 mychart-0.1.0.tgz 和 index.yaml 上传到 k8s-node1 的/var/www/charts目录，如图11-51所示。

```

root@k8s-node1:~#
root@k8s-node1:~# ls /var/www/charts/
index.yaml  mychart-0.1.0.tgz
root@k8s-node1:~#

```

图11-51

(5) 通过helm repo add将新仓库添加到Helm，如图11-52所示。

```

ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ helm repo add newrepo http://192.168.56.106:8080/charts
"newrepo" has been added to your repositories
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ helm repo list
NAME      URL
stable    https://kubernetes-charts.storage.googleapis.com
local     http://127.0.0.1:8879/charts
newrepo   http://192.168.56.106:8080/charts
ubuntu@k8s-master:~$

```

图11-52

仓库命名为newrepo，Helm会从仓库下载index.yaml。

(6) 现在已经可以repo search到mychart了，如图11-53所示。

```
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ helm search mychart
NAME          VERSION DESCRIPTION
local/mychart 0.1.0    A Helm chart for Kubernetes
newrepo/mychart 0.1.0    A Helm chart for Kubernetes
ubuntu@k8s-master:~$
```

图11-53

除了newrepo/mychart，这里还有一个local/mychart。这是因为在执行第2步打包操作的同时，mychart也被同步到了local的仓库。

(7) 已经可以直接从新仓库安装mychart了，如图11-54所示。

```
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ helm install newrepo/mychart
NAME:      guilded-jellyfish
LAST DEPLOYED: Sat Oct 21 16:49:31 2017
NAMESPACE: default
STATUS: DEPLOYED

RESOURCES:
==> v1/Service
NAME                CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
guilded-jellyfish-mychart 10.109.0.87 <none>        80/TCP    0s

==> v1beta1/Deployment
NAME                DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
guilded-jellyfish-mychart 1         1         1             0           0s

NOTES:
1. Get the application URL by running these commands:
  export POD_NAME=$(kubectl get pods --namespace default -l "app=mychart,r
  echo "Visit http://127.0.0.1:8080 to use your application"
  kubectl port-forward $POD_NAME 8080:80
```

图11-54

(8) 若以后仓库添加了新的chart，则需要用helm repo update更新本地的index，如图11-55所示。

```
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ helm repo update
Hang tight while we grab the latest from your chart repositories...
...Skip local chart repository
...Successfully got an update from the "newrepo" chart repository
...Successfully got an update from the "stable" chart repository
Update Complete. ✨ Happy Helming! ✨
ubuntu@k8s-master:~$
```

图11-55

这个操作相当于Ubutun的apt-get update。

11.6 小结

本章我们学习了Kubernetes包管理器Helm。

Helm让我们能够像apt管理deb包那样安装、部署、升级和删除容器化应用。

Helm由客户端和Tiller服务器组成。客户端负责管理chart，服务器负责管理release。

chart是Helm的应用打包格式，它由一组文件和目录构成。其中最重要的是模板，模板中定义了Kubernetes各类资源的配置信息，Helm在部署时通过values.yaml实例化模板。

Helm允许用户开发自己的chart，并为用户提供了调试工具。用户可以搭建自己的chart仓库，在团队中共享chart。

Helm帮助用户在Kubernetes上高效地运行和管理微服务架构应用，Helm非常重要。

第12章 网络

本章我们讨论Kubernetes网络这个重要主题。

Kubernetes作为编排引擎管理着分布在不同节点上的容器和Pod。Pod、Service、外部组件之间需要一种可靠的方式找到彼此并进行通信，Kubernetes网络则负责提供这个保障。本章包括如下内容：

- (1) Kubernetes网络模型。
- (2) 各种网络方案。
- (3) Network Policy。

12.1 Kubernetes网络模型

Kubernetes采用的是基于扁平地址空间的网络模型，集群中的每个Pod都有自己的IP地址，Pod之间不需要配置NAT就能直接通信。另外，同一个Pod中的容器共享Pod的IP，能够通过localhost通信。

这种网络模型对应用开发者和管理员相当友好，应用可以非常方便地从传统网络迁移到Kubernetes。每个Pod可被看作是一个个独立的系统，而Pod中的容器则可被看作同一系统中的不同进程。

下面讨论在这个网络模型下集群中的各种实体如何通信。知识点前面都已经涉及，这里可当作复习和总结。

1. Pod内容器之间的通信

当Pod被调度到某个节点，Pod中的所有容器都在这个节点上运行，这些容器共享相同的本地文件系统、IPC和网络命名空间。

不同Pod之间不存在端口冲突的问题，因为每个Pod都有自己的IP地址。当某个容器使用localhost时，意味着使用的是容器所属Pod的地址空间。

比如Pod A有两个容器container-A1和container-A2，container-A1在端口1234上监听，当container-A2连接到localhost:1234时，实际上就是在访问container-A1。这不会与同一个节点上的Pod B冲突，即使Pod B中的容器container-B1也在监听1234端口。

2. Pod之间的通信

Pod的IP是集群可见的，即集群中的任何其他Pod和节点都可以通过IP直接与Pod通信，这种通信不需要借助任何网络地址转换、隧道或代理技术。Pod内部和外部使用的是同一个IP，这也意味着标准的命名服务和发现机制，比如DNS可以直接使用。

3. Pod与Service的通信

Pod间可以直接通过IP地址通信，但前提是Pod知道对方的IP。在Kubernetes集群中，Pod可能会频繁地销毁和创建，也就是说Pod的IP

不是固定的。为了解决这个问题，**Service**提供了访问**Pod**的抽象层。无论后端的**Pod**如何变化，**Service**都作为稳定的前端对外提供服务。同时，**Service**还提供了高可用和负载均衡功能，**Service**负责将请求转发给正确的**Pod**。

4. 外部访问

无论是**Pod**的IP还是**Service**的Cluster IP，它们只能在Kubernetes集群中可见，对集群之外的世界，这些IP都是私有的。

Kubernetes提供了两种方式让外界能够与**Pod**通信：

- **NodePort**。Service通过Cluster节点的静态端口对外提供服务。外部可以通过<NodeIP>:<NodePort>访问Service。
- **LoadBalancer**。Service利用cloud provider提供的load balancer对外提供服务，cloud provider负责将load balancer的流量导向Service。目前支持的cloud provider有GCP、AWS、Azur等。

12.2 各种网络方案

网络模型有了，如何实现呢？

为了保证网络方案的标准化、扩展性和灵活性，Kubernetes采用了Container Networking Interface（CNI）规范。

CNI是由CoreOS提出的容器网络规范，使用了插件（Plugin）模型创建容器的网络栈，如图12-1所示。

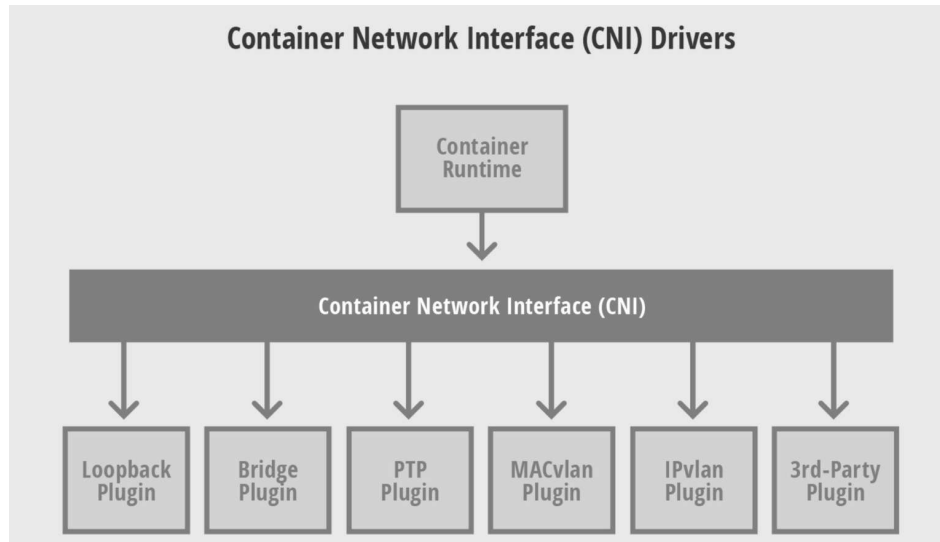


图12-1

CNI的优点是支持多种容器runtime，不仅仅是Docker。CNI的插件模型支持不同组织和公司开发的第三方插件，这对运维人员来说很有吸引力，可以灵活选择适合的网络方案。

目前已有多种支持Kubernetes的网络方案，比如Flannel、Calico、Canal、Weave Net等。因为它们都实现了CNI规范，用户无论选择哪种方案，得到的网络模型都一样，即每个Pod都有独立的IP，可以直接通信。区别在于不同方案的底层实现不同，有的采用基于VxLAN的Overlay实现，有的则是Underlay，性能上有区别。再有就是是否支持Network Policy。

12.3 Network Policy

Network Policy是Kubernetes的一种资源。Network Policy通过Label选择Pod，并指定其他Pod或外界如何与这些Pod通信。

默认情况下，所有Pod是非隔离的，即任何来源的网络流量都能够访问Pod，没有任何限制。当为Pod定义了Network Policy时，只有Policy允许的流量才能访问Pod。

不过，不是所有的Kubernetes网络方案都支持Network Policy。比如Flannel就不支持，Calico是支持的。我们接下来将用Canal来演示

Network Policy。Canal这个开源项目很有意思，它用Flannel实现Kubernetes集群网络，同时又用Calico实现Network Policy。

12.3.1 部署Canal

部署Canal与部署其他Kubernetes网络方案非常类似，都是在执行了kubeadm init初始化Kubernetes集群之后通过kubectl apply安装相应的网络方案。也就是说，没有太好的办法直接切换使用不同的网络方案，基本上只能重新创建集群。

要销毁当前集群，最简单的方法是在每个节点上执行kubeadm reset，然后就可以按照3.3.1小节“初始化Master”中的方法初始化集群了。

```
kubeadm init --apiserver-advertise-address 192.168.56.105
```

```
--pod-network-cidr=10.244.0.0/16
```

然后按照文档<https://kubernetes.io/docs/setup/independent/create-cluster-kubeadm/>安装Canal。文档列出了各种网络方案的安装方法，如图12-2所示。



图12-2

执行如下命令部署Canal:

```
kubectl apply -f
```

```
https://raw.githubusercontent.com/projectcalico/canal/master/k8s-install/1.7/rbac.yaml
```

```
kubectl apply -f
```

<https://raw.githubusercontent.com/projectcalico/canal/master/k8s-install/1.7/canal.yaml>

部署成功后，可以查看到Canal相关组件，如图12-3所示。

```
ubuntu@k8s-master:~$ kubectl get --namespace=kube-system daemonset canal
NAME          DESIRED  CURRENT  READY  UP-TO-DATE  AVAILABLE  NODE SELECTOR  AGE
canal         3        3        3      3           3          <none>         5d
ubuntu@k8s-master:~$ kubectl get --namespace=kube-system pod -o wide | grep canal
canal-fkrl8           3/3      Running  0       5d         192.168.56.107  k8s-node2
canal-pqvq2           3/3      Running  0       5d         192.168.56.105  k8s-master
canal-wtlhk           3/3      Running  0       5d         192.168.56.106  k8s-node1
ubuntu@k8s-master:~$
```

图12-3

Canal 作为 DaemonSet 部署到每个节点，属于 kube-system 这个 namespace。

12.3.2 实践Network Policy

为了演示 Network Policy，我们先部署一个 httpd 应用，其配置文件 httpd.yaml 如图 12-4 所示。

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: httpd
spec:
  replicas: 3
  template:
    metadata:
      labels:
        run: httpd
    spec:
      containers:
      - name: httpd
        image: httpd:latest
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 80

---
apiVersion: v1
kind: Service
metadata:
  name: httpd-svc
spec:
  type: NodePort
  selector:
    run: httpd
  ports:
  - protocol: TCP
    nodePort: 30000
    port: 8080
    targetPort: 80
```

图12-4

httpd有三个副本，通过NodePort类型的Service对外提供服务。部署应用，如图12-5所示。

```

ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl apply -f httpd.yml
deployment "httpd" created
service "httpd-svc" created
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get pod -o wide
NAME                READY   STATUS    RESTARTS   AGE   IP            NODE
httpd-b5c6f48-gs7p2 1/1     Running   0           2m    10.244.1.7    k8s-node1
httpd-b5c6f48-p86tv 1/1     Running   0           2m    10.244.1.8    k8s-node1
httpd-b5c6f48-qxzxg 1/1     Running   0           2m    10.244.2.4    k8s-node2
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get service httpd-svc
NAME      TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)          AGE
httpd-svc NodePort    10.104.77.239 <none>        8080:30000/TCP   2m
ubuntu@k8s-master:~$

```

图12-5

当前没有定义任何Network Policy，验证应用可以被访问，如图12-6所示。

(1) 启动一个busybox Pod，既可以访问Service，也可以Ping到副本Pod。

```

ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl run busybox --rm -ti --image=busybox /bin/sh
If you don't see a command prompt, try pressing enter.
/ #
/ # wget httpd-svc:8080
Connecting to httpd-svc:8080 (10.104.77.239:8080)
index.html      100% |*****|
/ #
/ # ping 10.244.1.7
PING 10.244.1.7 (10.244.1.7): 56 data bytes
64 bytes from 10.244.1.7: seq=0 ttl=62 time=1.696 ms
64 bytes from 10.244.1.7: seq=1 ttl=62 time=0.558 ms
64 bytes from 10.244.1.7: seq=2 ttl=62 time=0.497 ms

```

图12-6

(2) 集群节点既可以访问Service，也可以Ping到副本Pod，如图12-7所示。

```

root@k8s-node2:~#
root@k8s-node2:~# curl 10.104.77.239:8080
<html><body><h1>It works!</h1></body></html>
root@k8s-node2:~#
root@k8s-node2:~# ping -c 3 10.244.1.7
PING 10.244.1.7 (10.244.1.7) 56(84) bytes of data.
64 bytes from 10.244.1.7: icmp_seq=1 ttl=63 time=0.487 ms
64 bytes from 10.244.1.7: icmp_seq=2 ttl=63 time=0.494 ms
64 bytes from 10.244.1.7: icmp_seq=3 ttl=63 time=0.495 ms

```

图12-7

(3) 集群外 (192.168.56.1) 可以访问Service, 如图12-8所示。

```
→ ~  
→ ~ curl 192.168.56.106:30000  
<html><body><h1>It works!</h1></body></html>  
→ ~
```

图12-8

现在创建Network Policy, 如图12-9所示。

```
kind: NetworkPolicy  
apiVersion: networking.k8s.io/v1  
metadata:  
  name: access-httpd  
spec:  
  podSelector:  
    matchLabels:  
      run: httpd ①  
  ingress:  
    - from:  
      - podSelector:  
          matchLabels:  
            access: "true" ②  
      ports:  
        - protocol: TCP ③  
          port: 80
```

图12-9

① 定义将此Network Policy中的访问规则应用于label为run: httpd的Pod, 即httpd应用的三个副本Pod。

② ingress中定义只有label为access: "true"的Pod才能访问应用。

③ 只能访问80端口。

通过kubectl apply创建Network Policy, 如图12-10所示。

```

ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl apply -f policy.yaml
networkpolicy "access-httpd" created
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get networkpolicy
NAME                POD-SELECTOR  AGE
access-httpd        run=httpd     32s
ubuntu@k8s-master:~$

```

图12-10

验证Network Policy的有效性:

(1) busybox Pod已经不能访问Service，如图12-11所示。

```

ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl run busybox --rm -ti --image=busybox /bin/sh
If you don't see a command prompt, try pressing enter.
/ #
/ # wget httpd-svc:8080 --timeout=5
Connecting to httpd-svc:8080 (10.104.77.239:8080)
wget: download timed out
/ #

```

图12-11

如果Pod添加了label access: "true"就能访问到应用，但Ping已经被禁止，如图12-12所示。

```

ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl run busybox --rm -ti --labels="access=true" --image=busybox /bin/sh
If you don't see a command prompt, try pressing enter.
/ #
/ # wget httpd-svc:8080
Connecting to httpd-svc:8080 (10.104.77.239:8080)
index.html      100% |*****| 45  0:00:00
/ #
/ # ping -c 3 10.244.1.7
PING 10.244.1.7 (10.244.1.7): 56 data bytes

--- 10.244.1.7 ping statistics ---
3 packets transmitted, 0 packets received, 100% packet loss
/ #

```

图12-12

(2) 集群节点已经不能访问Service，也Ping不到副本Pod，如图12-13所示。

```
root@k8s-node2:~#  
root@k8s-node2:~# curl 10.104.77.239:8080 --connect-timeout 5  
curl: (28) Connection timed out after 5001 milliseconds  
root@k8s-node2:~#  
root@k8s-node2:~# ping -c 3 10.244.1.7  
PING 10.244.1.7 (10.244.1.7) 56(84) bytes of data.  
  
--- 10.244.1.7 ping statistics ---  
3 packets transmitted, 0 received, 100% packet loss, time 1999ms
```

图12-13

(3) 集群外 (192.168.56.1) 已经不能访问Service，如图12-14所示。

```
→ ~  
→ ~ curl 192.168.56.106:30000 --connect-timeout 5  
curl: (28) Connection timed out after 5003 milliseconds  
→ ~
```

图12-14

如果希望让集群节点和集群外 (192.168.56.1) 也能够访问到应用，可以对Network Policy做如图12-15所示的修改。

```
kind: NetworkPolicy  
apiVersion: networking.k8s.io/v1  
metadata:  
  name: access-httpd  
spec:  
  podSelector:  
    matchLabels:  
      run: httpd  
  ingress:  
    - from:  
      - podSelector:  
          matchLabels:  
            access: "true"  
      - ipBlock:  
          cidr: 192.168.56.0/24  
  ports:  
    - protocol: TCP  
      port: 80
```

图12-15

应用新的Network Policy，如图12-16所示。

```
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ kubectl apply -f policy.yaml  
networkpolicy "access-httpd" configured  
ubuntu@k8s-master:~$
```

图12-16

现在，集群节点和集群外（192.168.56.1）已经能够访问了，如图12-17、图12-18所示。

```
root@k8s-node2:~#  
root@k8s-node2:~# curl 10.104.77.239:8080  
<html><body><h1>It works!</h1></body></html>  
root@k8s-node2:~#
```

图12-17

```
→ ~  
→ ~ curl 192.168.56.106:30000  
<html><body><h1>It works!</h1></body></html>  
→ ~
```

图12-18

除了通过ingress限制进入的流量，也可以用egress限制外出的流量。大家可以参考官网相关文档和示例，这里就不赘述了。

12.4 小结

Kubernetes采用的是扁平化的网络模型，每个Pod都有自己的IP，并且可以直接通信。

CNI规范使得Kubernetes可以灵活选择多种Plugin实现集群网络。

Network Policy赋予了Kubernetes强大的网络访问控制机制。

第13章 Kubernetes Dashboard

前面章节Kubernetes所有的操作我们都是通过命令行工具kubect完成的。为了提供更丰富的用户体验，Kubernetes还开发了一个基于Web的Dashboard，用户可以用Kubernetes Dashboard部署容器化的应用、监控应用的状态、执行故障排查任务以及管理Kubernetes的各种资源。

在Kubernetes Dashboard中可以查看集群中应用的运行状态，也能够创建和修改各种Kubernetes资源，比如Deployment、Job、DaemonSet等。用户可以Scale Up/Down Deployment、执行Rolling Update、重启某个Pod或者通过向导部署新的应用。Dashboard能显示集群中各种资源的状态以及日志信息。

可以说，Kubernetes Dashboard提供了kubectl的绝大部分功能，大家可以根据情况进行选择。

13.1 安装

Kubernetes默认没有部署Dashboard，可通过如下命令安装：

```
kubect1 create -f https://raw.githubusercontent.com/kubernetes/dashboard/master/src/dep1oy/recommende1/kubernetes-dashboard.yaml
```

Dashboard会在kube-system namespace中创建自己的Deployment和服务，如图13-1所示。

```
ubuntu@k8s-master:~$ kubectl --namespace=kube-system get deployment kubernetes-dashboard
NAME                DESIRED    CURRENT    UP-TO-DATE    AVAILABLE    AGE
kubernetes-dashboard 1           1          1             1            17h
ubuntu@k8s-master:~$ kubectl --namespace=kube-system get service kubernetes-dashboard
NAME                TYPE        CLUSTER-IP    EXTERNAL-IP    PORT(S)    AGE
kubernetes-dashboard ClusterIP    10.110.199.111 <none>         443/TCP    17h
```

图13-1

因为Service是ClusterIP类型，为了方便使用，我们可通过kubect1 --namespace=kube-system edit service kubernetes-dashboard 修改成

NodePort类型，如图13-2所示。

```
spec:
  clusterIP: 10.110.199.111
  ports:
  - port: 443
    protocol: TCP
    targetPort: 8443
  selector:
    k8s-app: kubernetes-dashboard
  sessionAffinity: None
  type: NodePort
status:
  loadBalancer: {}
```

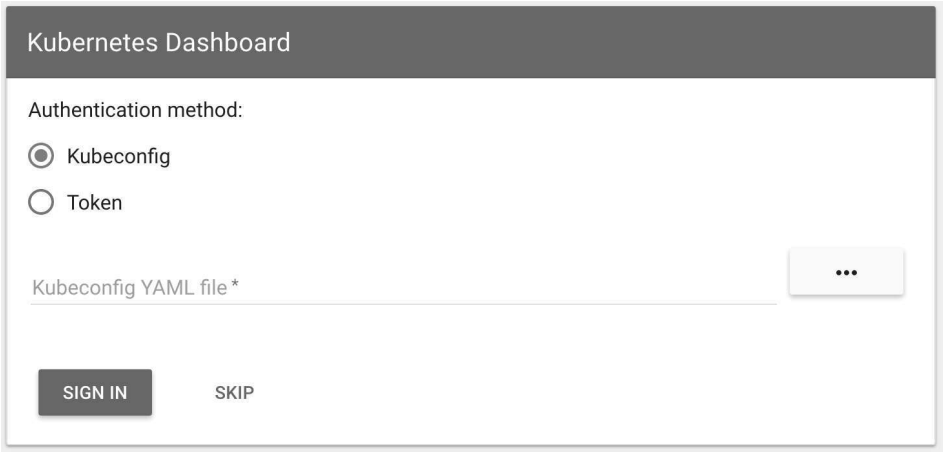
图13-2

保存修改，此时已经为Service分配了端口31614，如图13-3所示。

```
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl --namespace=kube-system get service kubernetes-dash
NAME                TYPE        CLUSTER-IP    EXTERNAL-IP    PORT(S)
kubernetes-dashboard NodePort     10.110.199.111 <none>         443:31614/TCP
ubuntu@k8s-master:~$
```

图13-3

通过浏览器访问Dashboard <https://192.168.56.105:31614/>，登录界面如图13-4所示。



The image shows the login page of the Kubernetes Dashboard. At the top, it says 'Kubernetes Dashboard'. Below that, under 'Authentication method:', there are two radio buttons: 'Kubeconfig' (which is selected) and 'Token'. Below the radio buttons is a text input field labeled 'Kubeconfig YAML file *' with a dropdown menu icon (three dots) to its right. At the bottom, there are two buttons: 'SIGN IN' and 'SKIP'.

图13-4

13.2 配置登录权限

Dashboard支持Kubeconfig和Token两种认证方式，为了简化配置，我们通过配置文件dashboard-admin.yaml为Dashboard默认用户赋予admin权限，如图13-5所示。

```
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRoleBinding
metadata:
  name: kubernetes-dashboard
  labels:
    k8s-app: kubernetes-dashboard
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
- kind: ServiceAccount
  name: kubernetes-dashboard
  namespace: kube-system
```

图13-5

执行kubectl apply使之生效，如图13-6所示。

```
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl apply -f dashboard-admin.yaml
clusterrolebinding "kubernetes-dashboard" created
ubuntu@k8s-master:~$
```

图13-6

现在直接单击登录页面中的SKIP就可以进入Dashboard了，如图13-7所示。

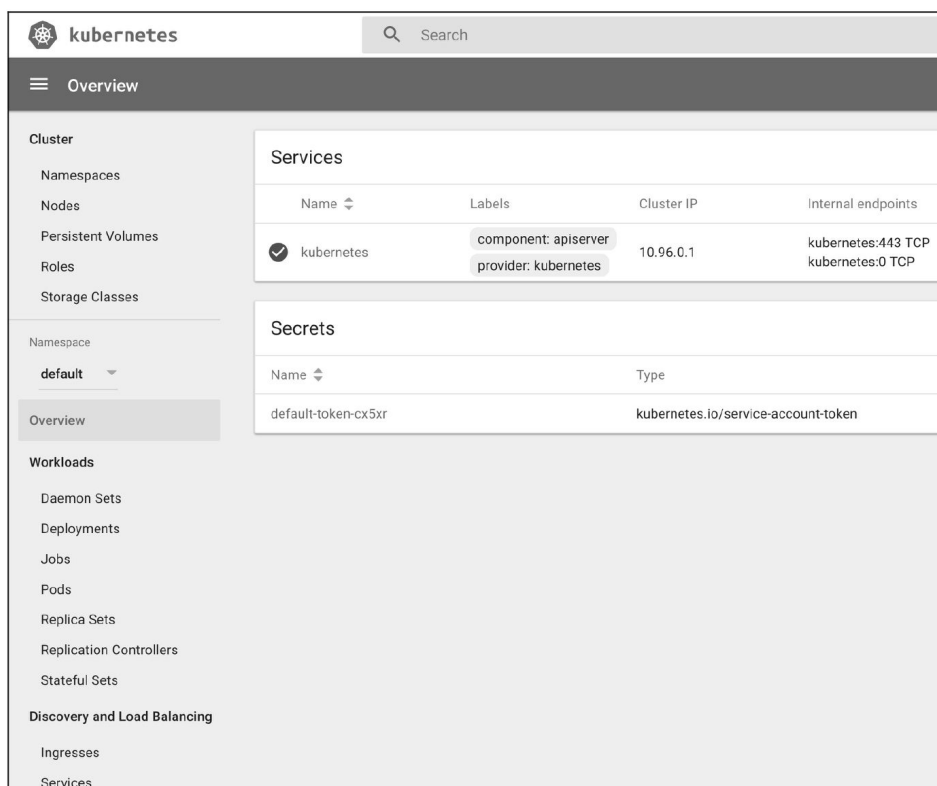


图13-7

13.3 Dashboard界面结构

Dashboard的界面很简洁，分为三个大的区域。

(1) 顶部操作区，如图13-8所示。在这里用户可以搜索集群中的资源、创建资源或退出。



图13-8

(2) 左边导航菜单。通过导航菜单可以查看和管理集群中的各种资源。菜单项按照资源的层级分为以下两类：

- Cluster级别的资源，如图13-9所示。

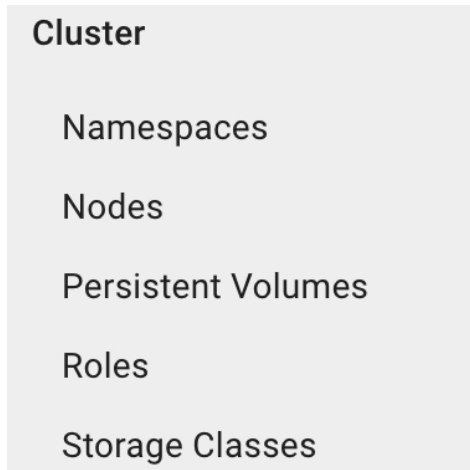


图13-9

- Namespace级别的资源，如图13-10所示。

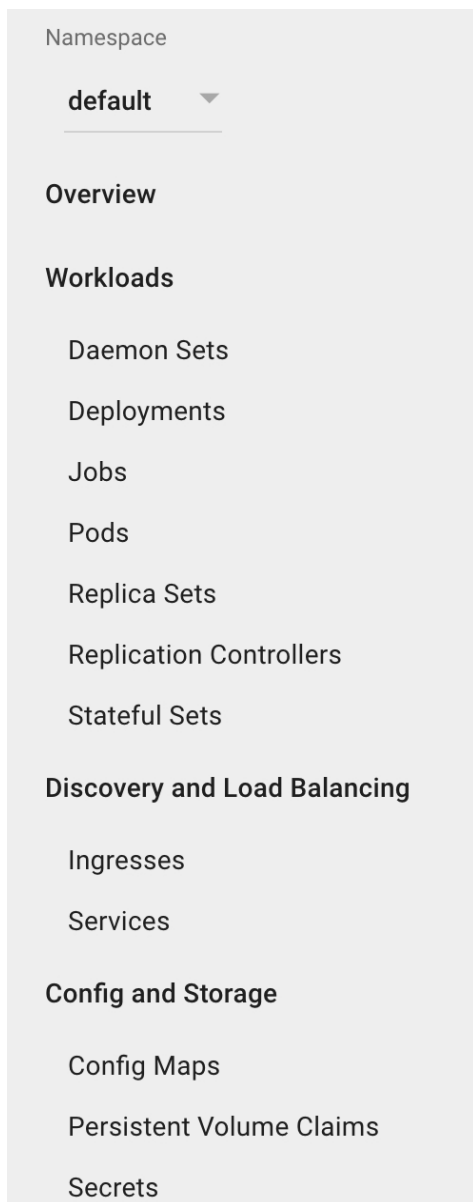


图13-10

默认显示的是default Namespace， 可以进行切换， 如图13-11所示。

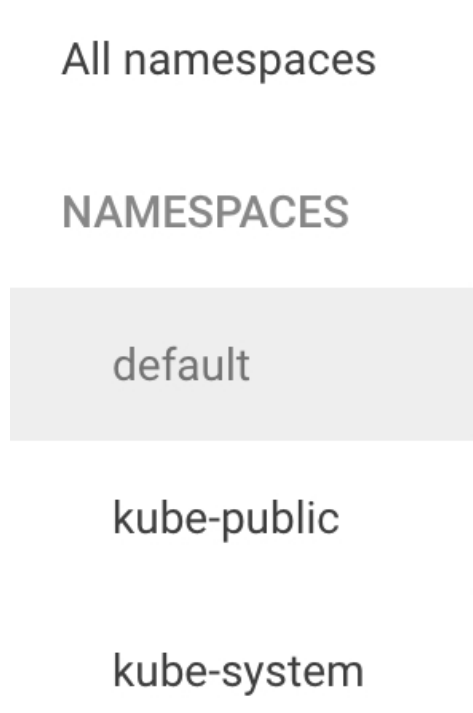


图13-11

(3) 中间主体区。

在导航菜单中单击了某类资源，中间主体区就会显示该资源所有实例，比如单击**Pods**，如图13-12所示。

Pods				
Name ↕	Node	Status ↕	Restarts	Age ↕
✓ kubernetes-dashboard-747c4f7cf-wwlz7	k8s-node1	Running	0	18 hours
✓ kube-proxy-jch75	k8s-node1	Running	0	18 hours
✓ kube-flannel-ds-f7wgk	k8s-node1	Running	0	18 hours
✓ kube-flannel-ds-lnjqt	k8s-node2	Running	0	18 hours
✓ kube-proxy-9cggh	k8s-node2	Running	0	18 hours
✓ kube-flannel-ds-bgclx	k8s-master	Running	0	18 hours
✓ kube-apiserver-k8s-master	k8s-master	Running	0	18 hours
✓ kube-scheduler-k8s-master	k8s-master	Running	0	18 hours
✓ etcd-k8s-master	k8s-master	Running	0	18 hours
✓ kube-controller-manager-k8s-master	k8s-master	Running	0	18 hours
✓ kube-dns-545bc4bfd4-6zr5w	k8s-master	Running	0	18 hours
✓ kube-proxy-q6j8m	k8s-master	Running	0	18 hours

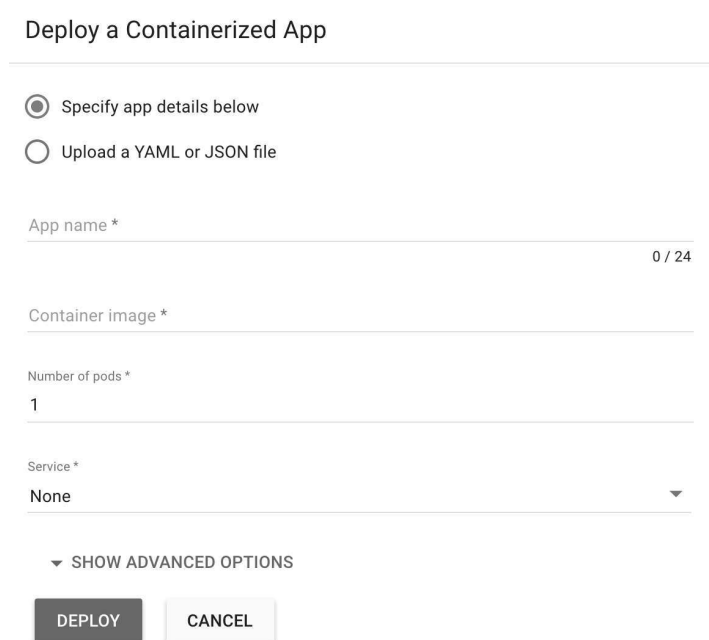
图13-12

13.4 典型使用场景

接下来我们介绍几个Dashboard的典型使用场景。

13.4.1 部署Deployment

单击顶部操作区的+CREATE按钮，如图13-13所示。



The screenshot shows a form titled "Deploy a Containerized App". It has two radio buttons: "Specify app details below" (selected) and "Upload a YAML or JSON file". Below the radio buttons are four input fields: "App name *" with a character count "0 / 24", "Container image *", "Number of pods *" with the value "1", and "Service *" with a dropdown menu showing "None". At the bottom, there is a link "SHOW ADVANCED OPTIONS" and two buttons: "DEPLOY" and "CANCEL".

图13-13

用户可以直接输入要部署应用的名字、镜像、副本数等信息，也可以上传YAML配置文件。如果是上传配置文件，则可以创建任意类型的资源，而不仅仅是Deployment。

13.4.2 在线操作

对于每种资源，都可以单击按钮执行各种操作，如图13-14所示。

Deployments						Search	×
Name	Namespace	Labels	Pods	Age	Images		
✓ kubelet-dashboard	kube-system	k8s-app: kubelet-das	1 / 1	18 hours	gcr.io/google_containers/k		
✓ kube-dns	kube-system	k8s-app: kube-dns	1 / 1	18 hours	gcr.io/google_containe gcr.io/google_containe gcr.io/google_containe	Scale	
						Delete	
						View/edit YAML	

图13-14

例如，单击View/edit YAML，可直接修改资源的配置，保存后立即生效，其效果与kubectl edit一样，如图13-15所示。

Edit a Deployment

1234567891011121314151617181920212223242526272829

```
{
  "kind": "Deployment",
  "apiVersion": "extensions/v1beta1",
  "metadata": {
    "name": "kube-dns",
    "namespace": "kube-system",
    "selfLink": "/apis/extensions/v1beta1/namespaces/kube-system
/deployments/kube-dns",
    "uid": "a676a020-c2c9-11e7-abb6-0800274451ad",
    "resourceVersion": "33294",
    "generation": 1,
    "creationTimestamp": "2017-11-06T08:08:19Z",
    "labels": {
      "k8s-app": "kube-dns"
    },
    "annotations": {
      "deployment.kubernetes.io/revision": "1"
    }
  },
  "spec": {
    "replicas": 1,
    "selector": {
      "matchLabels": {
        "k8s-app": "kube-dns"
      }
    },
    "template": {
      "metadata": {
        "creationTimestamp": null,
        "labels": {
```

CANCEL

COPY

UPDATE

图13-15

13.4.3 查看资源详细信息

单击某个资源实例的名字，可以查看详细信息，其效果与 `kubectl describe` 一样，如图13-16所示。

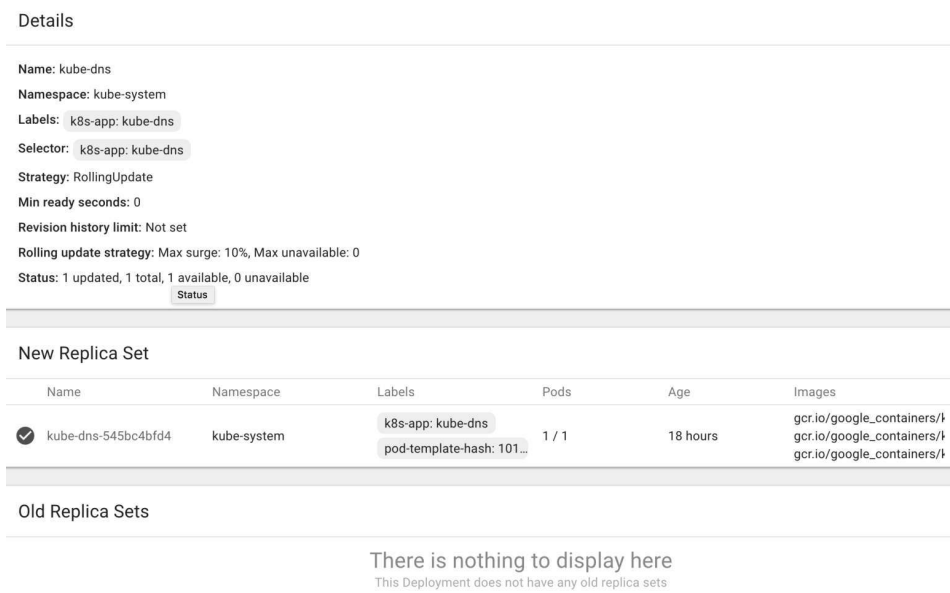


图13-16

13.4.4 查看Pod日志

在Pod及其父资源（DaemonSet、ReplicaSet等）页面单击  按钮，可以查看Pod的日志，其效果与 `kubectl logs` 一样，如图13-17所示。

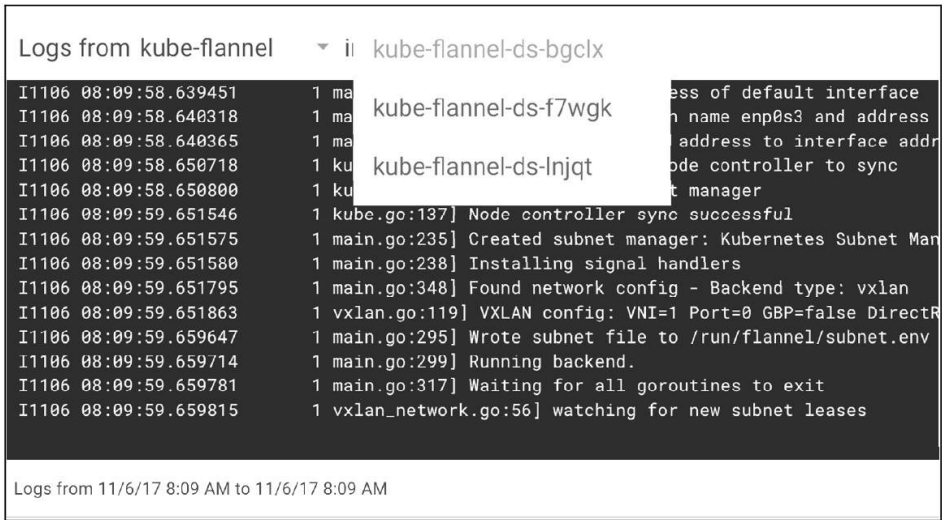


图13-17

Kubernetes Dashboard界面设计友好，自解释性强，可以看作GUI版的kubectl，更多功能留给大家自己探索。

13.5 小结

本章介绍了Kubernetes Dashboard的安装和使用方法。Dashboard能完成日常管理的大部分工作，可以作为命令行工具kubectl的有益补充。

第14章 Kubernetes集群监控

创建Kubernetes集群并部署容器化应用只是第一步。一旦集群运行起来，我们需要确保集群一起都是正常的，所有必要组件就位并各司其职，有足够的资源满足应用的需求。Kubernetes是一个复杂系统，运维团队需要有一套工具帮助他们获知集群的实时状态，并为故障排查提供及时和准确的数据支持。

本章重点讨论Kubernetes常用的监控方案，下一章会讨论日志管理。

14.1 Weave Scope

Weave Scope是Docker和Kubernetes可视化监控工具。Scope提供了自上而下的集群基础设施和应用的完整视图，用户可以轻松对分布式的容器化应用进行实时监控和问题诊断。

14.1.1 安装Scope

安装Scope的方法很简单，执行如下命令：

```
kubectl apply --namespace kube-system -f
    "https://cloud.weave.works/k8s/scope.yaml?k8s-
version=$(kubectl version | base64 | tr -d '\n')"
```

部署成功后，有如图14-1所示的相关组件。

```

ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get --namespace=kube-system daemonset weave-scope-agent
NAME          DESIRED  CURRENT  READY  UP-TO-DATE  AVAILABLE  NODE SELECTOR
weave-scope-agent 3         3        3      3            3          <none>
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get --namespace=kube-system deployment weave-scope-app
NAME          DESIRED  CURRENT  UP-TO-DATE  AVAILABLE  AGE
weave-scope-app 1         1        1            1          18h
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get --namespace=kube-system service weave-scope-app
NAME          TYPE        CLUSTER-IP  EXTERNAL-IP  PORT(S)          AGE
weave-scope-app  NodePort   10.106.149.68  <none>       80:30693/TCP     18h
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get --namespace=kube-system pod | grep weave
weave-scope-agent-765g2                1/1      Running    0          18h
weave-scope-agent-nchjr                1/1      Running    0          18h
weave-scope-agent-wzn4                 1/1      Running    0          18h
weave-scope-app-567cfdb6d5-kk2cg       1/1      Running    0          18h
ubuntu@k8s-master:~$

```

图14-1

(1) DaemonSet weave-scope-agent，集群每个节点上都会运行的scope agent程序，负责收集数据。

(2) Deployment weave-scope-app，scope应用，从agent获取数据，通过Web UI展示并与用户交互。

(3) Service weave-scope-app，默认是ClusterIP类型，为了方便，已通过kubectl edit修改为NodePort。

14.1.2 使用Scope

浏览器访问<http://192.168.56.106:30693/>，Scope默认显示当前所有的Controller（Deployment、DaemonSet等），如图14-2所示。

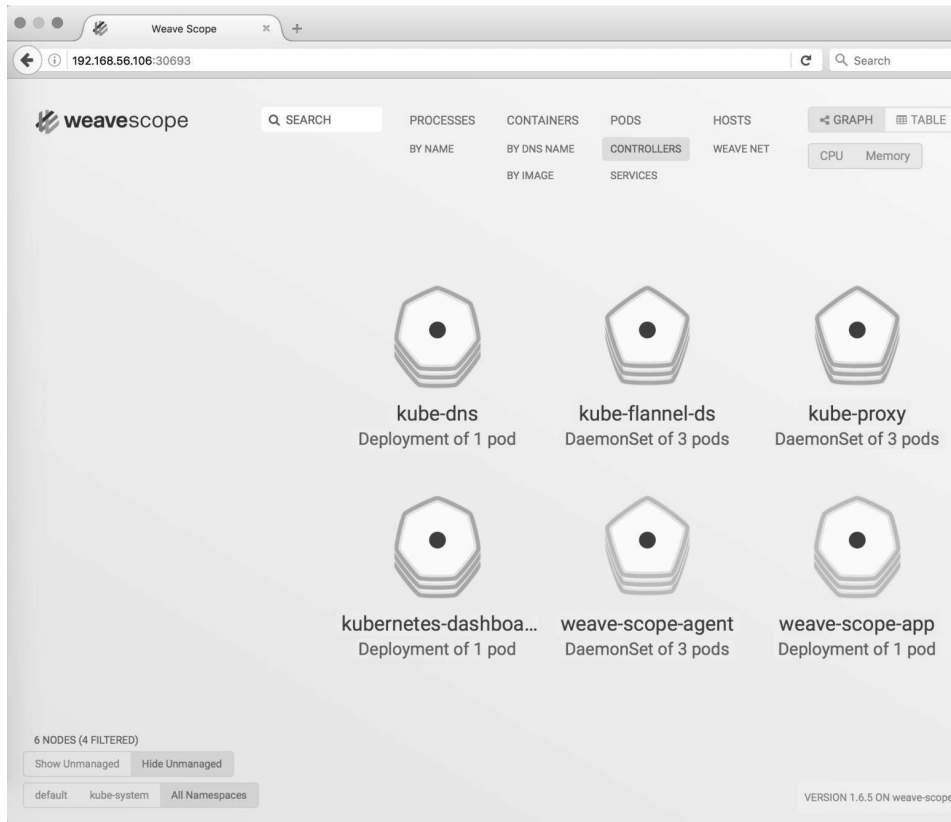


图14-2

1. 拓扑结构

Scope会自动构建应用和集群的逻辑拓扑，比如单击顶部PODS，会显示所有Pod以及Pod之间的依赖关系，如图14-3所示。

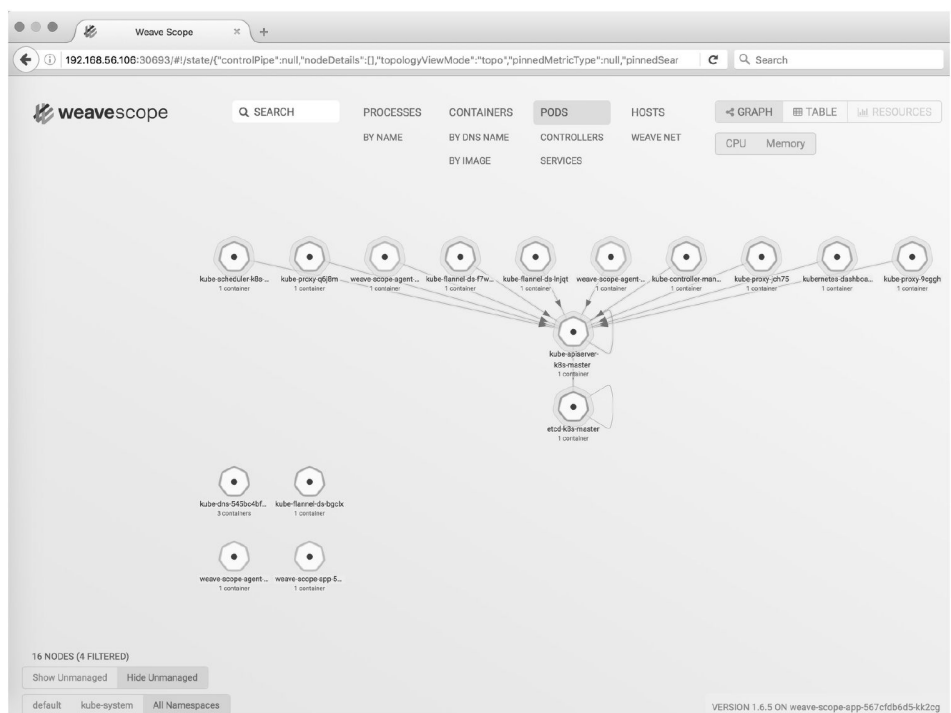


图14-3

单击HOSTS，会显示各个节点之间的关系，如图14-4所示。

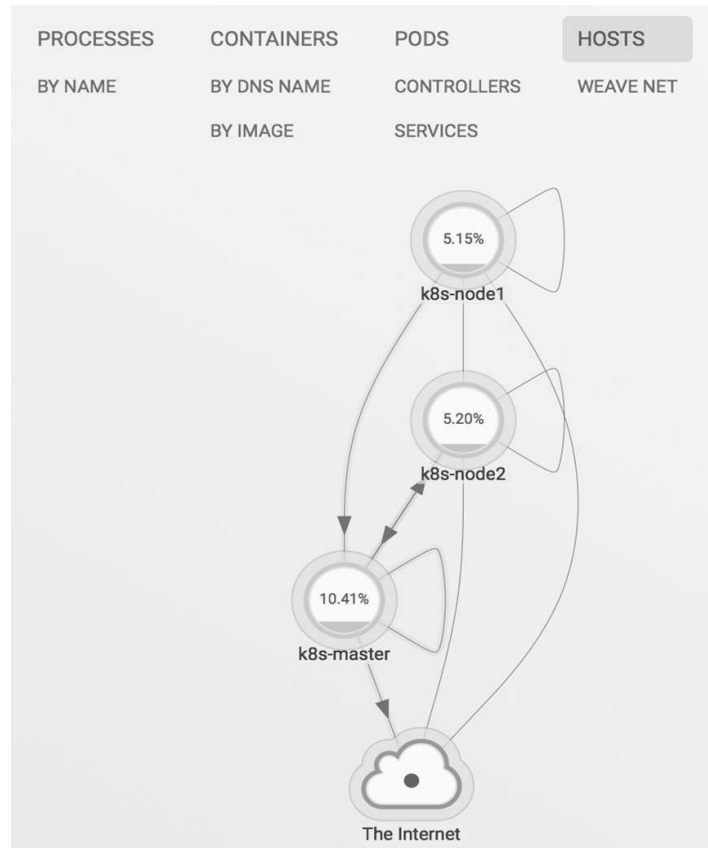


图14-4

2. 实时资源监控

可以在Scope中查看资源的CPU和内存使用情况，如图14-5所示。



图14-5

支持的资源有Host、Pod和Container，如图14-6、图14-7所示。

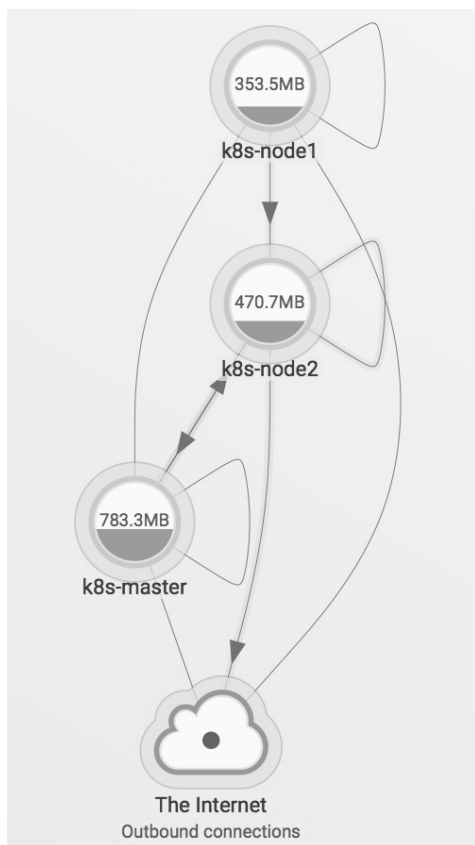


图14-6

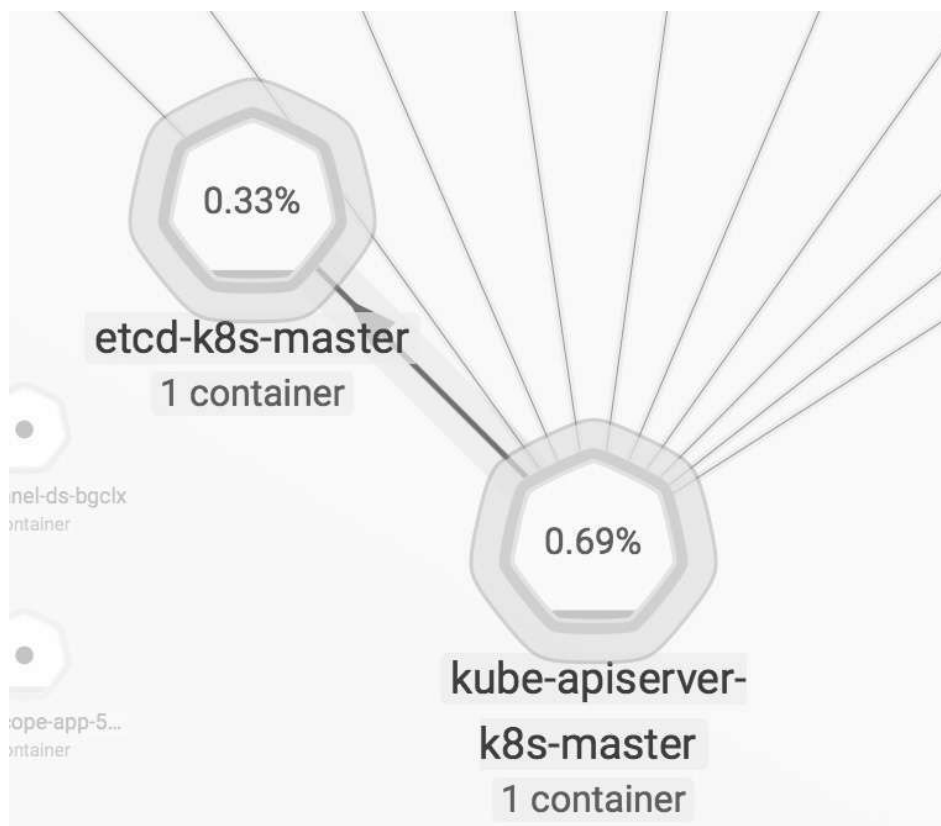


图14-7

3. 在线操作

Scope还提供了便捷的在线操作功能，比如选中某个Host，单击>_按钮可以直接在浏览器中打开节点的命令行终端，如图14-8所示。

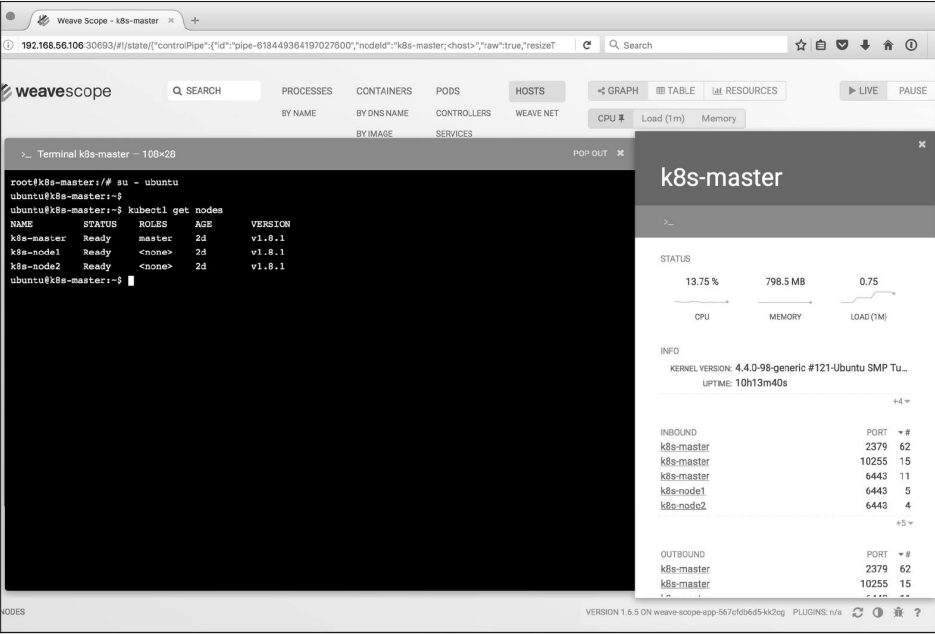


图14-8

单击Deployment的+可以执行Scale Up操作，如图14-9所示。

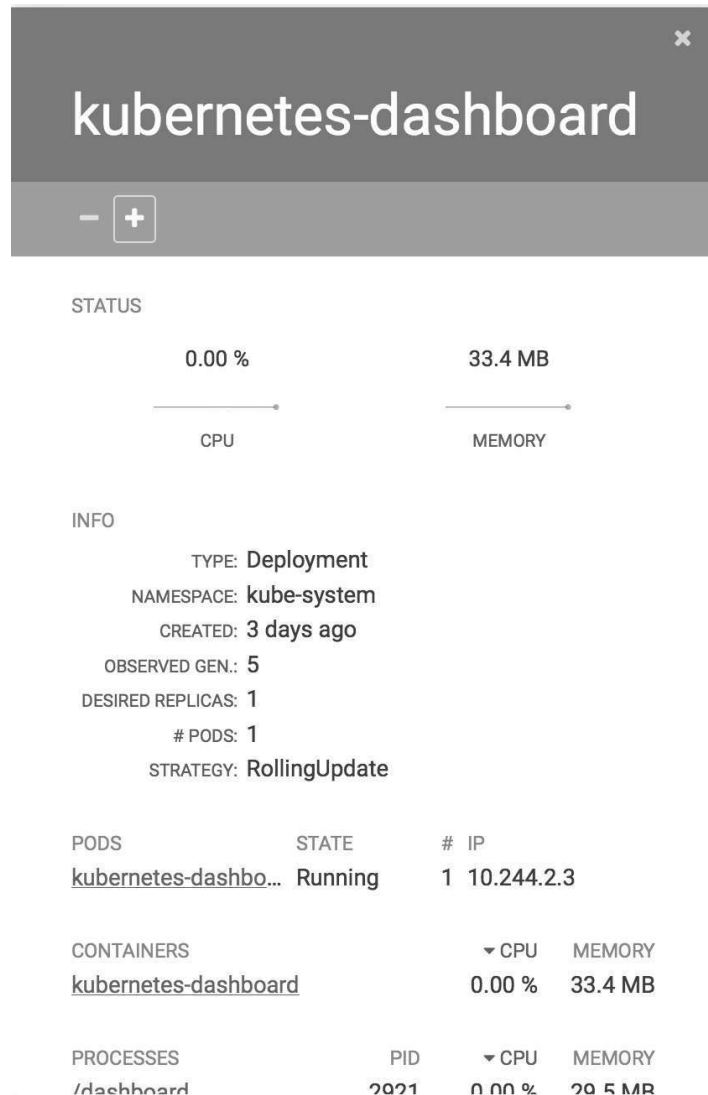


图14-9

查看Pod的日志，如图14-10所示。

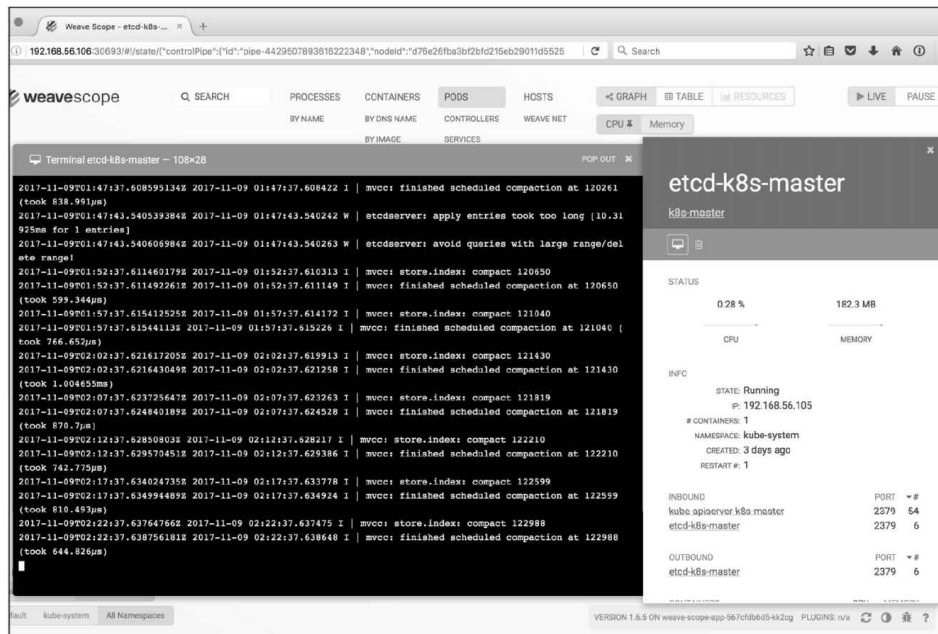


图14-10

可以查看attach、restart、stop容器，以及直接在Scope中排查问题，如图14-11所示。



图14-11

4. 强大的搜索功能

Scope支持关键字搜索和定位资源，如图14-12所示。还可以进行条件搜索，比如查找和定位MEMORY大于100MB的Pod，如图14-13所示。

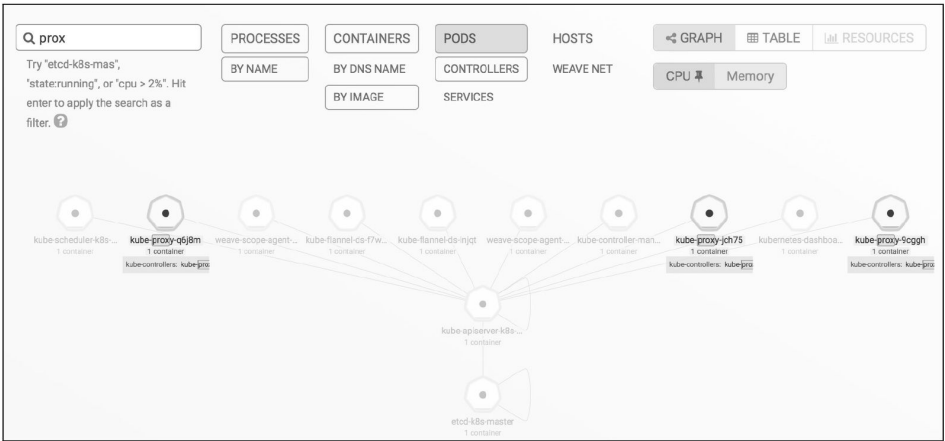


图14-12

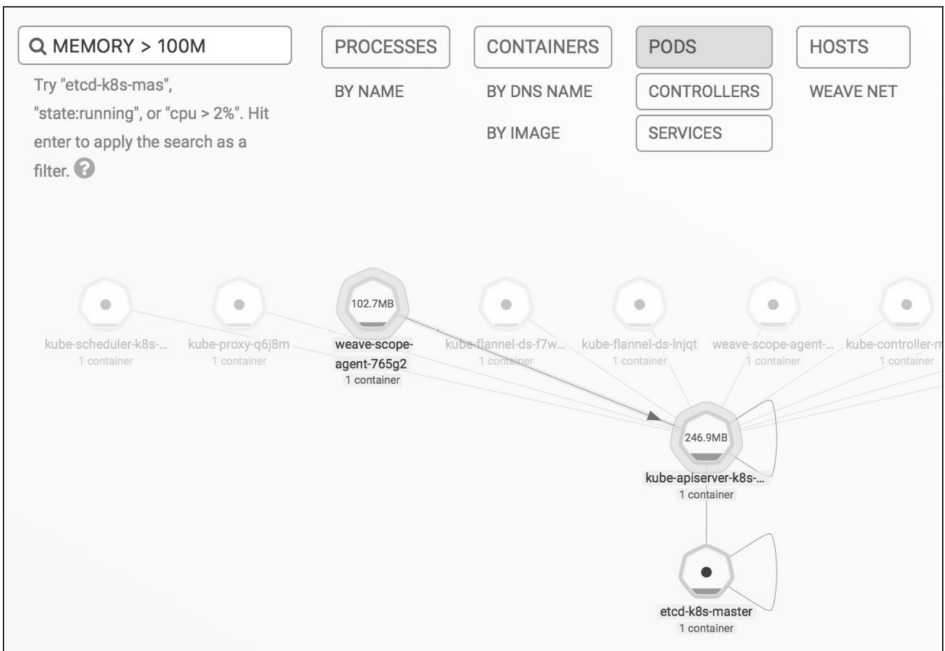


图14-13

Weave Scope界面极其友好，操作简洁流畅，更多功能留给大家去探索。

14.2 Heapster

Heapster是Kubernetes原生的集群监控方案。Heapster以Pod的形式运行，它会自动发现集群节点，从节点上的Kubelet获取监控数据。Kubelet则是从节点上的cAdvisor收集数据。

Heapster将数据按照Pod进行分组，将它们存储到预先配置的backend并进行可视化展示。Heapster当前支持的backend有InfluxDB（通过Grafana展示）、Google Cloud Monitoring等。Heapster的整体架构如图14-14所示。

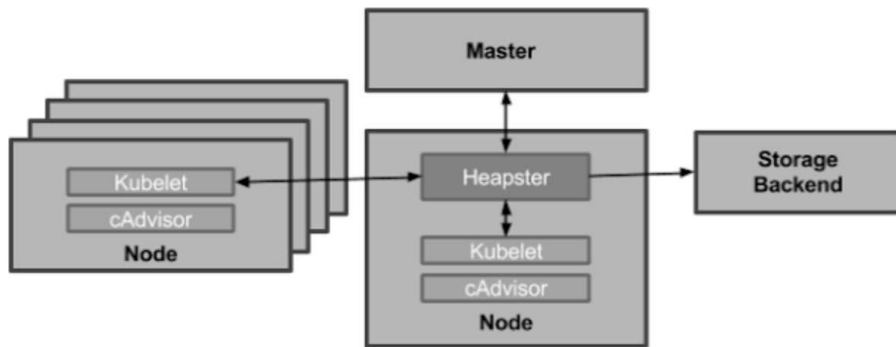


图14-14

下面我们将实践由Heapster、InfluxDB和Grafana组成的监控方案。Kubelet和cAdvisor是Kubernetes的自带组件，无须额外部署。

14.2.1 部署

Heapster本身是一个Kubernetes应用，部署方法很简单，运行如下命令：

```
git clone https://github.com/kubernetes/heapster.git
```

```
kubectl apply -f heapster/deploy/kube-config/influxdb/
```

```
kubectl apply -f heapster/deploy/kube-config/rbac/heapster-rbac.yaml
```

Heapster相关资源如图14-15所示。

```

ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get --namespace=kube-system deployment | grep -e heapster -e monitor
heapster                1                1                1                1                8m
monitoring-grafana      1                1                1                1                8m
monitoring-influxdb     1                1                1                1                8m
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get --namespace=kube-system service | grep -e heapster -e monitor
heapster                ClusterIP        10.108.228.4     <none>           80/TCP           8m
monitoring-grafana      NodePort         10.111.8.115     <none>           80:32314/TCP     8m
monitoring-influxdb     ClusterIP        10.99.44.147     <none>           8086/TCP         8m
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get --namespace=kube-system pod -o wide | grep -e heapster -e monitor
heapster-5d67855584-5hdqn    1/1            Running    0                8m            10.244.1.18    k8s-node2
monitoring-grafana-5bccc9f786-5bbkw    1/1            Running    0                8m            10.244.2.13    k8s-node1
monitoring-influxdb-85cb4985d4-t2b26    1/1            Running    0                8m            10.244.2.14    k8s-node1
ubuntu@k8s-master:~$

```

图14-15

为了便于访问，已通过kubectl edit将Service monitoring-grafana的类型修改为NodePort。

14.2.2 使用

浏览器打开Grafana的Web UI: <http://192.168.56.105:32314/>。

Heapster已经预先配置好了Grafana的DataSource和Dashboard，如图14-16所示。

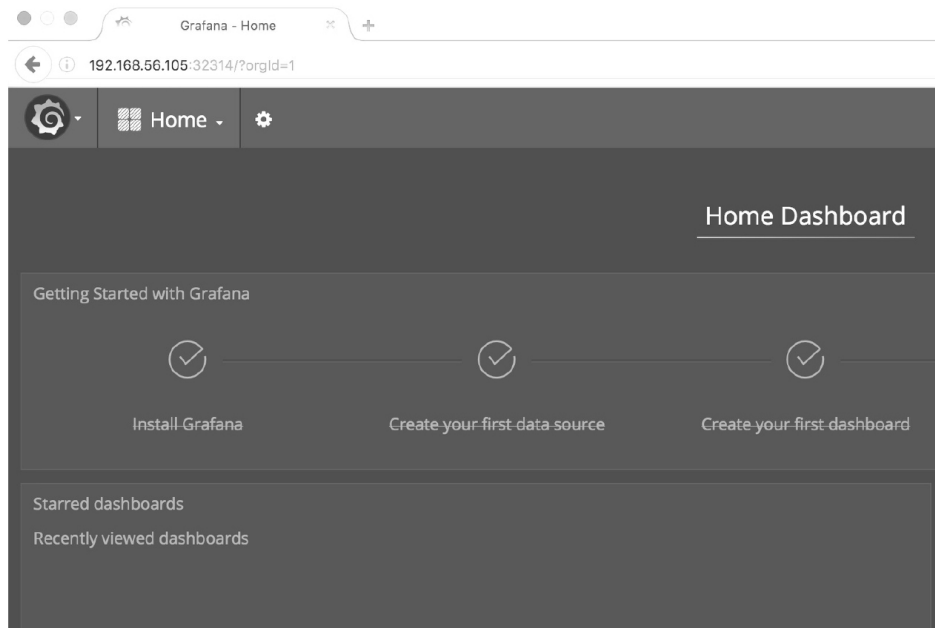


图14-16

单击左上角的**Home**菜单，可以看到预定义的**Dashboard Cluster**和**Pods**，如图14-17所示。

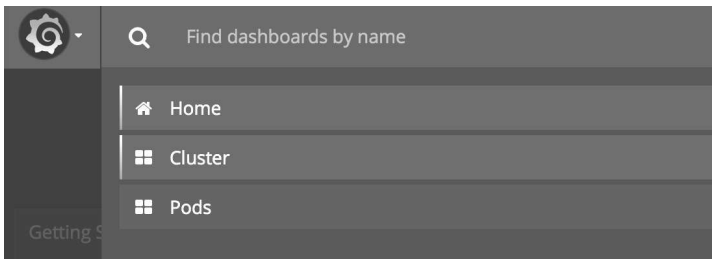


图14-17

单击**Cluster**，可以查看集群中节点的CPU、内存、网络 and 磁盘的使用情况，如图14-18所示。

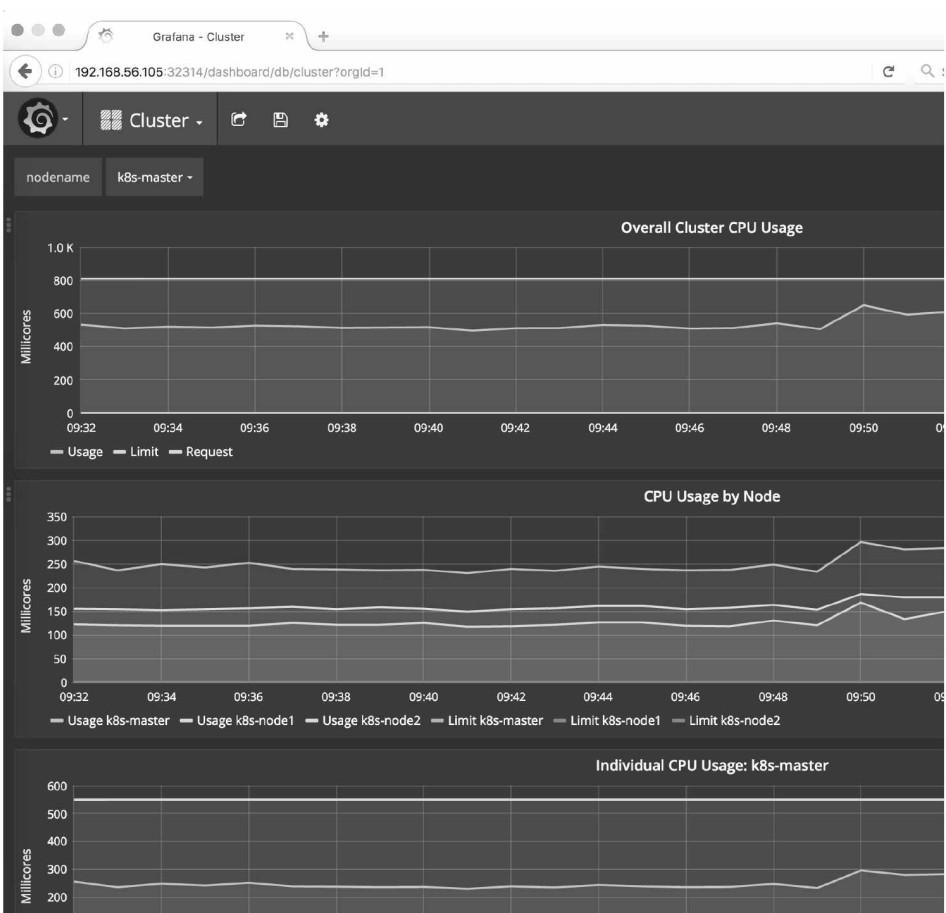


图14-18

在左上角可以切换查看不同节点的数据，如图14-19所示。

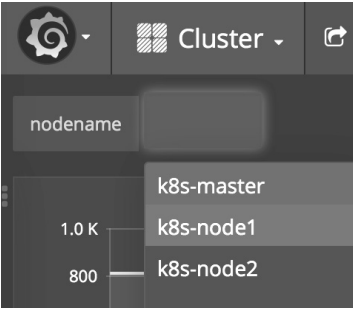


图14-19

切换到Pods Dashboard，可以查看Pod的监控数据，包括单个Pod的CPU、内存、网络和磁盘使用情况，如图14-20所示。

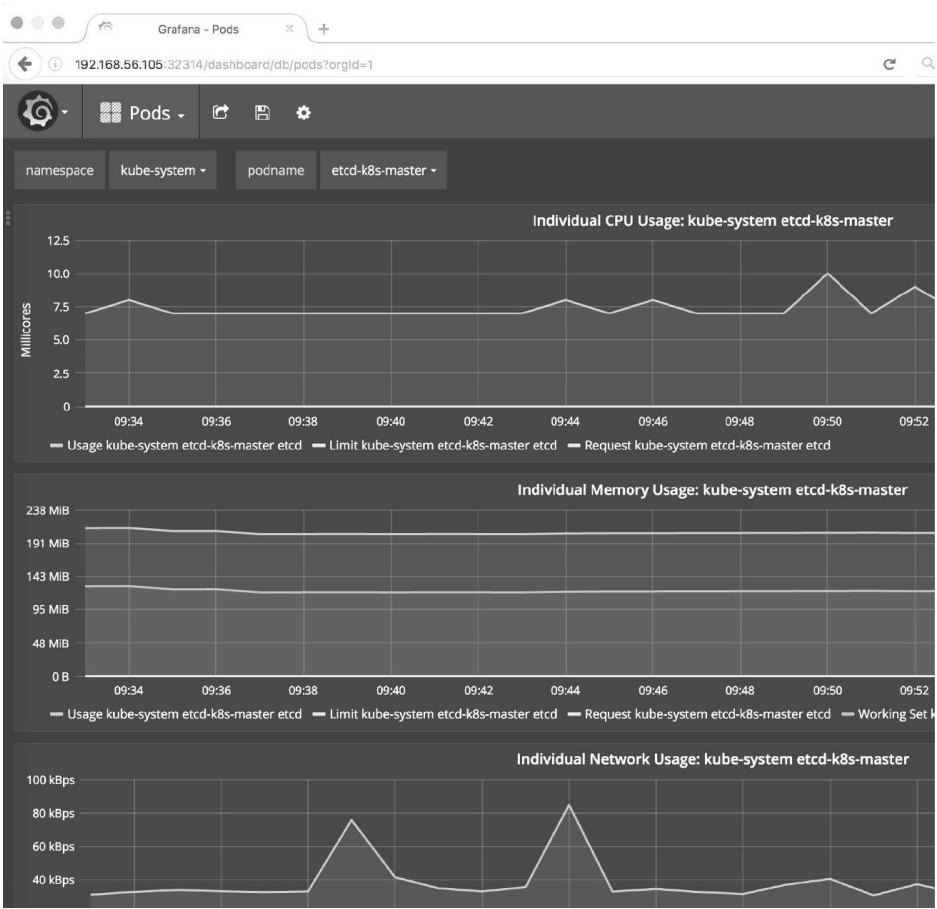


图14-20

在左上角可以切换到不同Namespace的Pod，如图14-21所示。

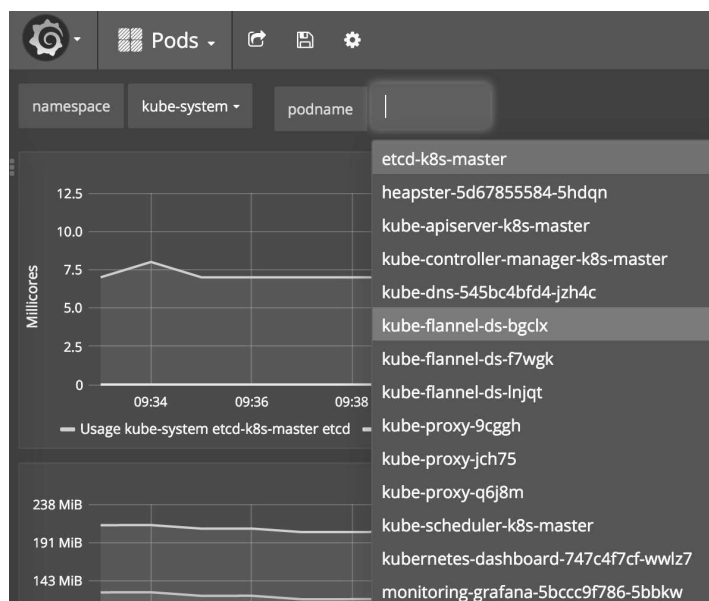


图14-21

Heapster预定义的Dashboard很直观，也很简单。如有必要，可以在Grafana中定义自己的Dashboard，满足特定的业务需求。

14.3 Prometheus Operator

前面我们介绍了Kubernetes的两种监控方案，即Weave Scope和Heapster，它们主要的监控对象是Node和Pod。这些数据对Kubernetes运维人员是必需的，但还不够。我们通常还希望监控集群本身的运行状态，比如Kubernetes的API Server、Scheduler、Controller Manager等管理组件是否正常工作以及负荷是否过大等。

本节我们将学习监控方案Prometheus Operator，它能回答上面这些问题。

Prometheus Operator是CoreOS开发的基于Prometheus的Kubernetes监控方案，也可能是目前功能最全面的开源方案。我们先通过截图了解一下它能干什么。

Prometheus Operator通过Grafana展示监控数据，预定义了一系列的Dashboard，如图14-22所示。

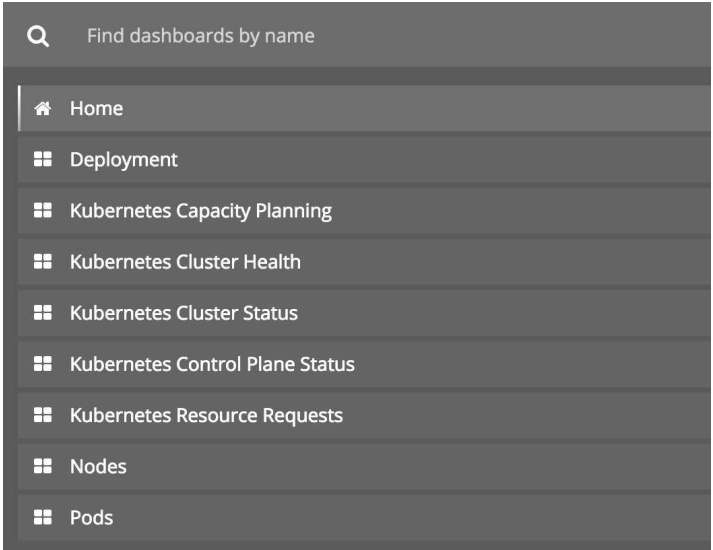


图14-22

- Kubernetes集群的整体健康状态如图14-23所示。

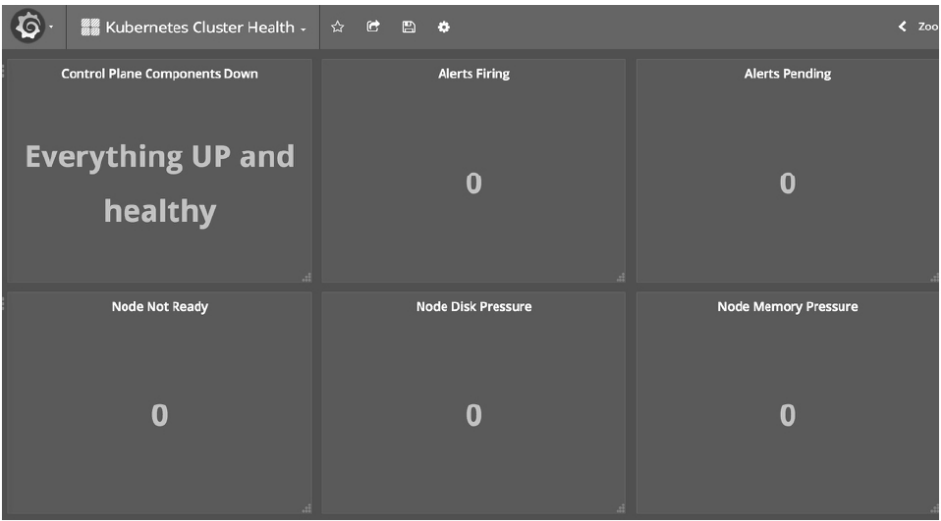


图14-23

- 整个集群的资源使用情况如图14-24、图14-25所示。



图14-24

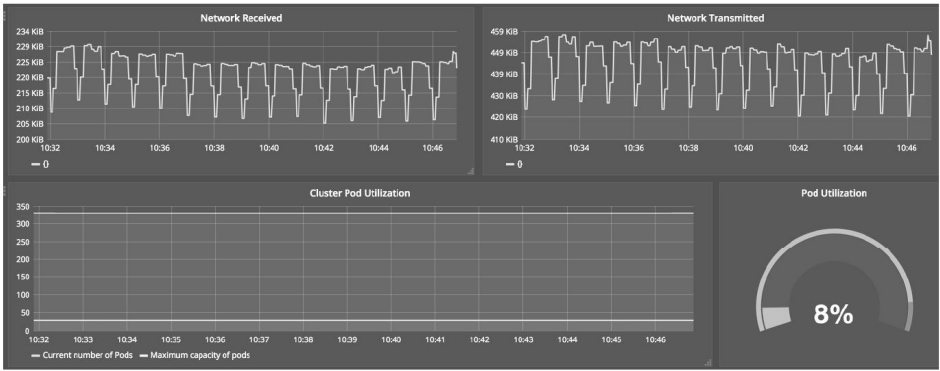


图14-25

- Kubernetes各个管理组件的状态如图14-26、图14-27所示。

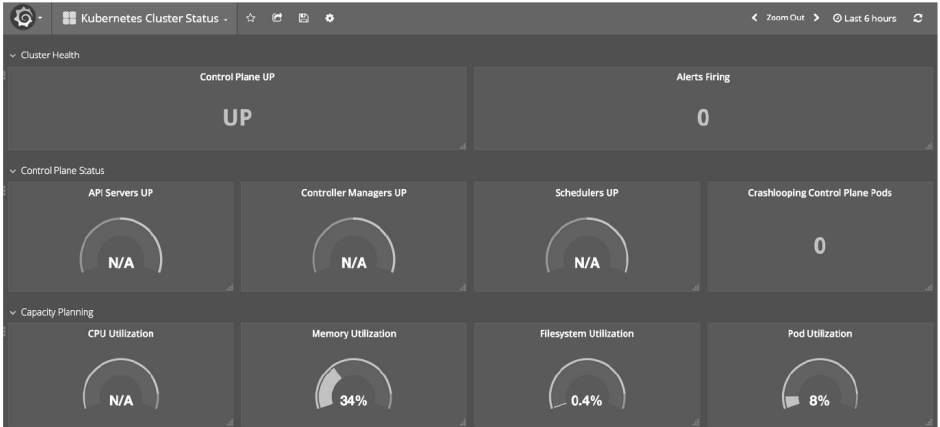


图14-26

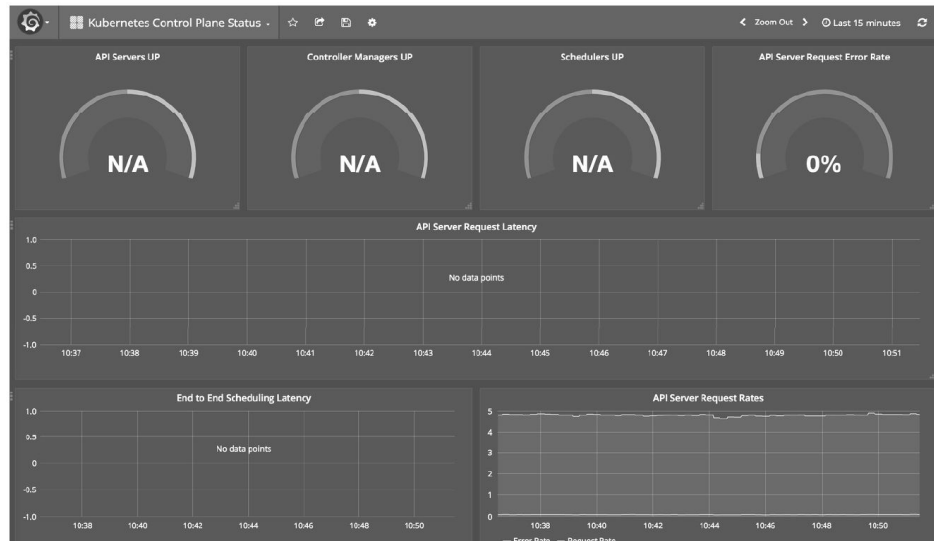


图14-27

- 节点的资源使用情况如图14-28所示。

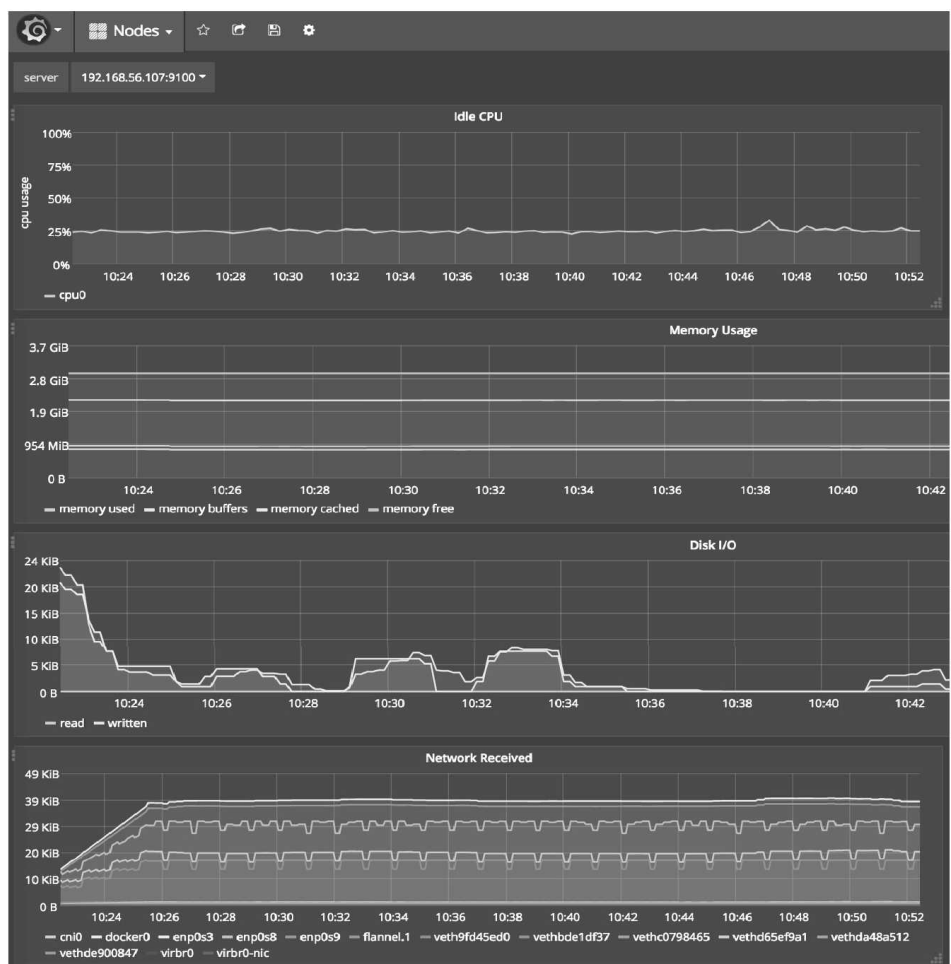


图14-28 (a)



图14-28 (b)

- Deployment的运行状态如图14-29所示。

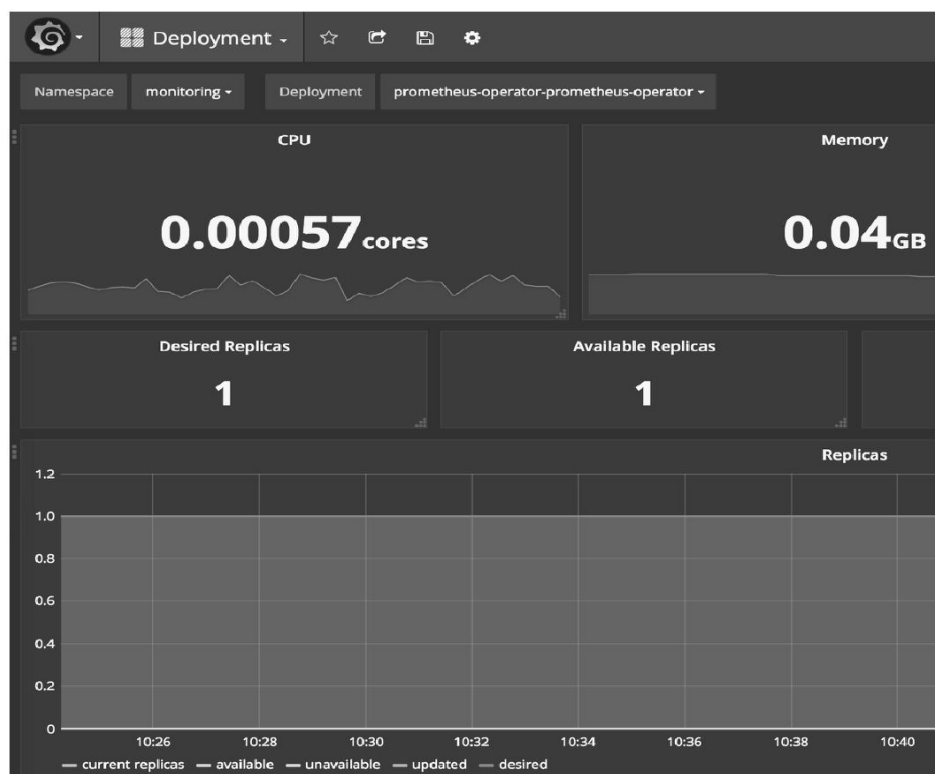


图14-29

- Pod的运行状态如图14-30所示。



图14-30

这些Dashboard展示了从集群到Pod的运行状况，能够帮助用户更好地运维Kubernetes，而且Prometheus Operator迭代非常快，相信会继续开发出更多更好的功能，所以值得我们花些时间学习和实践。

14.3.1 Prometheus架构

因为Prometheus Operator是基于Prometheus的，所以我们需要先了解一下Prometheus。

Prometheus是一个非常优秀的监控工具。准确地说应该是监控方案。Prometheus提供了数据搜集、存储、处理、可视化和告警一套完整的

解决方案。Prometheus的架构如图14-31所示。

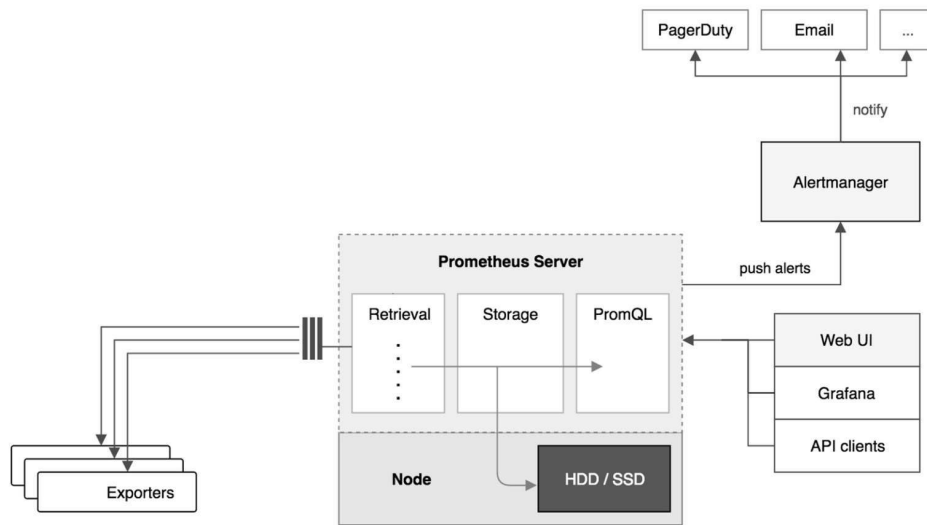


图14-31

官网上的原始架构图比上面这张要复杂一些，为了避免注意力分散，这里只保留了最重要的组件。

1. Prometheus Server

Prometheus Server负责从Exporter拉取和存储监控数据，并提供一套灵活的查询语言（PromQL）供用户使用。

2. Exporter

Exporter负责收集目标对象（host、container等）的性能数据，并通过HTTP接口供Prometheus Server获取。

3. 可视化组件

监控数据的可视化展现对于监控方案至关重要。以前Prometheus自己开发了一套工具，不过后来废弃了，因为开源社区出现了更为优秀的产品Grafana。Grafana能够与Prometheus无缝集成，提供完美的数据展示能力。

4. Alertmanager

用户可以定义基于监控数据的告警规则，规则会触发告警。一旦Alertmanager收到告警，就会通过预定义的方式发出告警通知，支持的方式包括Email、PagerDuty、Webhook等。

14.3.2 Prometheus Operator架构

Prometheus Operator的目标是尽可能简化在Kubernetes中部署和维护Prometheus的工作。其架构如图14-32所示。

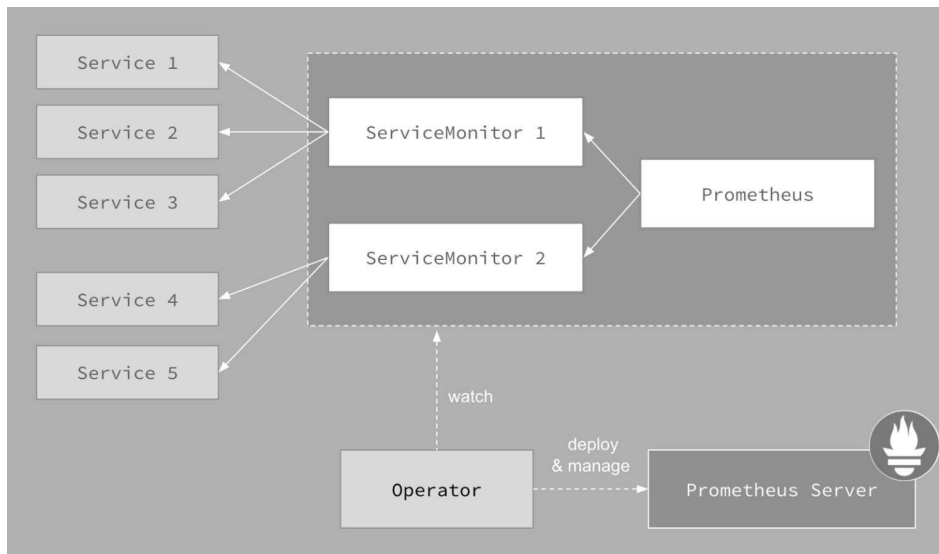


图14-32

图14-32中的每一个对象都是Kubernetes中运行的资源。

1. Operator

Operator即Prometheus Operator，在Kubernetes中以Deployment运行。其职责是部署和管理Prometheus Server，根据ServiceMonitor动态更新Prometheus Server的监控对象。

2. Prometheus Server

Prometheus Server会作为Kubernetes应用部署到集群中。为了更好地在Kubernetes中管理Prometheus，CoreOS的开发人员专门定义了一个命名

为Prometheus类型的Kubernetes定制化资源。我们可以把Prometheus看作一种特殊的Deployment，它的用途就是专门部署Prometheus Server。

3. Service

这里的Service就是Cluster中的Service资源，也是Prometheus要监控的对象，在Prometheus中叫作Target。每个监控对象都有一个对应的Service。比如要监控Kubernetes Scheduler，就得有一个与Scheduler对应的Service。当然，Kubernetes集群默认是没有这个Service的，Prometheus Operator会负责创建。

4. ServiceMonitor

Operator能够动态更新Prometheus的Target列表，ServiceMonitor就是Target的抽象。比如想监控Kubernetes Scheduler，用户可以创建一个与Scheduler Service相映射的ServiceMonitor对象。Operator则会发现这个新的ServiceMonitor，并将Scheduler的Target添加到Prometheus的监控列表中。

ServiceMonitor也是Prometheus Operator专门开发的一种Kubernetes定制化资源类型。

5. Alertmanager

除了Prometheus和ServiceMonitor，Alertmanager是Operator开发的第三种Kubernetes定制化资源。我们可以把Alertmanager看作一种特殊的Deployment，它的用途就是专门部署Alertmanager组件。

14.3.3 部署Prometheus Operator

笔者在实践时使用的是Prometheus Operator最新版本v0.14.0。由于项目开发迭代速度很快，部署方法可能会更新，必要时请参考官方文档。

1. 下载最新源码

```
git clone https://github.com/coreos/prometheus-operator.git
```

```
cd prometheus-operator
```

为方便管理，创建一个单独的Namespace monitoring，Prometheus Operator相关的组件都会部署到这个Namespace。

```
kubectl create namespace monitoring
```

2. 安装Prometheus Operator Deployment

```
helm install --name prometheus-operator --set rbacEnable=true--  
namespace=monitoring helm/prometheus-operator
```

Prometheus Operator所有的组件都打包成Helm Chart，安装部署非常方便，如图14-33所示。如果对Helm不熟悉，可以参考前面相关的章节。

```
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ kubectl get --namespace=monitoring deployment prometheus-operator  
NAME                                DESIRED   CURRENT   UP-TO-DATE   AVAILABLE  
prometheus-operator-prometheus-operator 1         1         1            1  
ubuntu@k8s-master:~$
```

图14-33

3. 安装Prometheus、Alertmanager和Grafana

```
helm          install          --name          prometheus          --  
set           serviceMonitorsSelector.app=prometheus--  
set           ruleSelector.app=prometheus          --  
namespace=monitoring helm/prometheus
```

```
helm          install          --name          alertmanager          --  
namespace=monitoring helm/alertmanager
```

```
helm install --name grafana --namespace=monitoring helm/grafana
```

可以通过kubectl get prometheus查看Prometheus类型的资源，如图14-34所示。

```

ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get --namespace=monitoring prometheus
NAME          AGE
prometheus    1d
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get --namespace=monitoring pod prometheus-prometheus-0
NAME                READY   STATUS    RESTARTS   AGE
prometheus-prometheus-0  2/2     Running   0           1d
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get --namespace=monitoring service prometheus-prometheus
NAME                TYPE        CLUSTER-IP    EXTERNAL-IP  PORT(S)          AGE
prometheus-prometheus  NodePort    10.96.207.169 <none>       9090:30413/TCP  1d
ubuntu@k8s-master:~$

```

图14-34

为了方便访问Prometheus Server，这里已经将Service类型通过kubectl edit改为NodePort。

同样可以查看Alertmanager和Grafana的相关资源，如图14-35、图14-36所示。

```

ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get --namespace=monitoring alertmanager
NAME          AGE
alertmanager  2d
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get --namespace=monitoring pod alertmanager-alertmanager-0
NAME                READY   STATUS    RESTARTS   AGE
alertmanager-alertmanager-0  2/2     Running   0           2d
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get --namespace=monitoring service alertmanager-alertmanager
NAME                TYPE        CLUSTER-IP    EXTERNAL-IP  PORT(S)          AGE
alertmanager-alertmanager  NodePort    10.103.53.199 <none>       9093:32758/TCP  2d
ubuntu@k8s-master:~$

```

图14-35

```

ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get --namespace=monitoring deployment grafana-grafana
NAME                DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
grafana-grafana     1          1          1             1           5h
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get --namespace=monitoring pod grafana-grafana-5fdf676c68-v7dlb
NAME                READY   STATUS    RESTARTS   AGE
grafana-grafana-5fdf676c68-v7dlb  2/2     Running   0           5h
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get --namespace=monitoring service grafana-grafana
NAME                TYPE        CLUSTER-IP    EXTERNAL-IP  PORT(S)          AGE
grafana-grafana     NodePort    10.109.107.156 <none>       80:32342/TCP  5h
ubuntu@k8s-master:~$

```

图14-36

Service类型也都已经改为NodePort。

4. 安装kube-prometheus

kube-prometheus是一个Helm Chart，打包了监控Kubernetes需要的所有Exporter和ServiceMonitor。

```
helm install --name kube-prometheus --namespace=monitoring helm/kube-prometheus
```

每个Exporter会对应一个Service，为Pormetheus提供Kubernetes集群的各类监控数据，如图14-37所示。

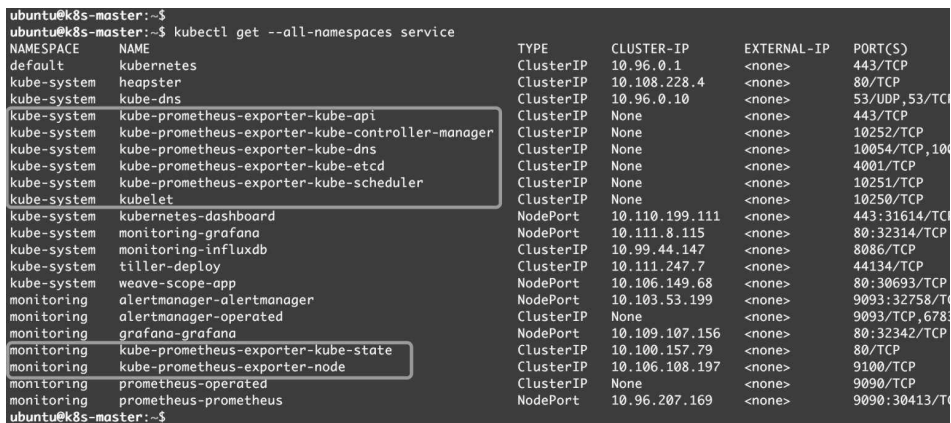


图14-37

每个Service对应一个ServiceMonitor，组成Pormetheus的Target列表，如图14-38所示。

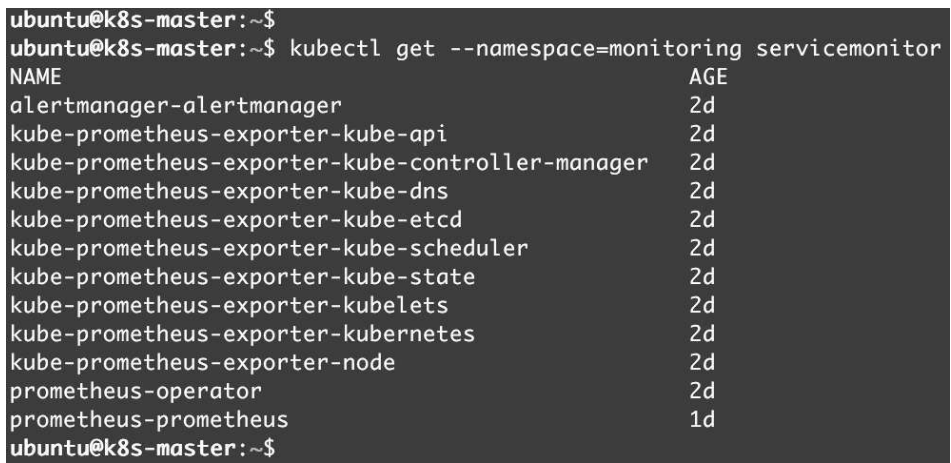


图14-38

与Prometheus Operator相关的所有Pod如图14-39所示。

```

ubuntu@k8s-master:~$ kubectl get pods --namespace=monitoring -o wide
NAME                                READY   STATUS    RESTARTS   AGE   IP            NODE
alertmanager-alertmanager-0         2/2    Running   0           2d    10.244.1.26   k8s-node2
grafana-grafana-5fdf676c68-v7dlb    2/2    Running   0           5h    10.244.1.30   k8s-node2
kube-prometheus-exporter-kube-state-5ff8596498-lprrr 2/2    Running   0           2d    10.244.2.31   k8s-node1
kube-prometheus-exporter-node-6hb77 1/1    Running   0           2d    192.168.56.106 k8s-node1
kube-prometheus-exporter-node-k59dm 1/1    Running   0           2d    192.168.56.107 k8s-node2
kube-prometheus-exporter-node-t4cns 1/1    Running   0           2d    192.168.56.105 k8s-master
prometheus-operator-prometheus-operator-597f678b79-94qvw 1/1    Running   1           2d    10.244.2.27   k8s-node1
prometheus-prometheus-0             2/2    Running   0           1d    10.244.2.32   k8s-node1
ubuntu@k8s-master:~$

```

图14-39

我们注意到有些Exporter没有运行Pod，这是因为像API Server、Scheduler、Kubelet等Kubernetes内部组件原生就支持Prometheus，只需要定义Service就能直接从预定义端口获取监控数据。

通过浏览器打开 Prometheus 的 Web UI (<http://192.168.56.105:30413/targets>)，如图14-40所示。

Endpoint	State	Labels
alertmanager		
http://10.244.1.26:9093/metrics	UP	endpoint="http" instance="10.244.1.26:9093" namespace="monitoring" pod="alertmanager-alertmanager-0" service="alertmanager-alertmanager"
kube-prometheus-exporter-kube-dns		
http://10.244.1.21:10054/metrics	UP	endpoint="http-metrics-dnames" instance="10.244.1.21:10054" namespace="kube-system" pod="kube-dns-5" service="kube-prometheus-exporter-kube-dns"
http://10.244.1.21:10055/metrics	UP	endpoint="http-metrics-skydns" instance="10.244.1.21:10055" namespace="kube-system" pod="kube-dns-54t" service="kube-prometheus-exporter-kube-dns"
kube-prometheus-exporter-kube-state		
http://10.244.2.31:8080/metrics	UP	endpoint="kube-state-metrics" instance="10.244.2.31:8080" namespace="monitoring" pod="kube-prometheus-exporter-kube-state-5ff8596498-lprrr" service="kube-prometheus-exporter-kube-state"
kube-prometheus-exporter-node		
http://192.168.56.105:9100/metrics	UP	endpoint="metrics" instance="192.168.56.105:9100" namespace="monitoring" pod="kube-prometheus-exporter-node" service="kube-prometheus-exporter-node"
http://192.168.56.106:9100/metrics	UP	endpoint="metrics" instance="192.168.56.106:9100" namespace="monitoring" pod="kube-prometheus-exporter-node" service="kube-prometheus-exporter-node"

图14-40

可以看到所有Target的状态都是UP。

5. 安装Alert规则

Prometheus Operator提供了默认的Alertmanager告警规则，通过如下命令安装。

```
sed -ie 's/role: prometheus-  
rulefiles/app: prometheus/g' contrib/kube-  
prometheus/manifests/prometheus/prometheus-k8s-rules.yaml
```

```
sed -  
ie 's/prometheus: k8s/prometheus: prometheus/g' contrib/kube-  
prometheus/manifests/prometheus/prometheus-k8s-rules.yaml
```

```
sed -ie 's/job=\"kube-controller-manager/job=\"kube-prometheus-  
exporter-kube-controller-manager/g' contrib/kube-  
prometheus/manifests/prometheus/prometheus-k8s-rules.yaml
```

```
sed -ie 's/job=\"apiserver/job=\"kube-prometheus-exporter-kube-  
api/g' contrib/kube-prometheus/manifests/prometheus/prometheus-  
k8s-rules.yaml
```

```
sed -ie 's/job=\"kube-scheduler/job=\"kube-prometheus-exporter-  
kube-scheduler/g' contrib/kube-  
prometheus/manifests/prometheus/prometheus-k8s-rules.yaml
```

```
sed -ie 's/job=\"node-exporter/job=\"kube-prometheus-exporter-  
node/g' contrib/kube-  
prometheus/manifests/prometheus/prometheus-k8s-rules.yaml
```

```
kubectl apply -n monitoring -f contrib/kube-  
prometheus/manifests/prometheus/prometheus-k8s-rules.yaml
```

6. 安装Grafana Dashboard

Prometheus Operator定义了显示监控数据的默认Dashboard，通过如下命令安装。

```
sed -ie 's/grafana-dashboards-0/grafana-  
grafana/g' contrib/kube-prometheus/manifests/grafana/grafana-  
dashboards.yaml
```

```
sed -ie 's/prometheus-k8s.monitoring/prometheus-prometheus.monitoring/g' contrib/kube-prometheus/manifests/grafana/grafana-dashboards.yaml
```

```
kubectl apply -n monitoring -f contrib/kube-prometheus/manifests/grafana/grafana-dashboards.yaml
```

打开 Grafana 的 Web UI (<http://192.168.56.105:32342/>)，如图 14-41 所示。

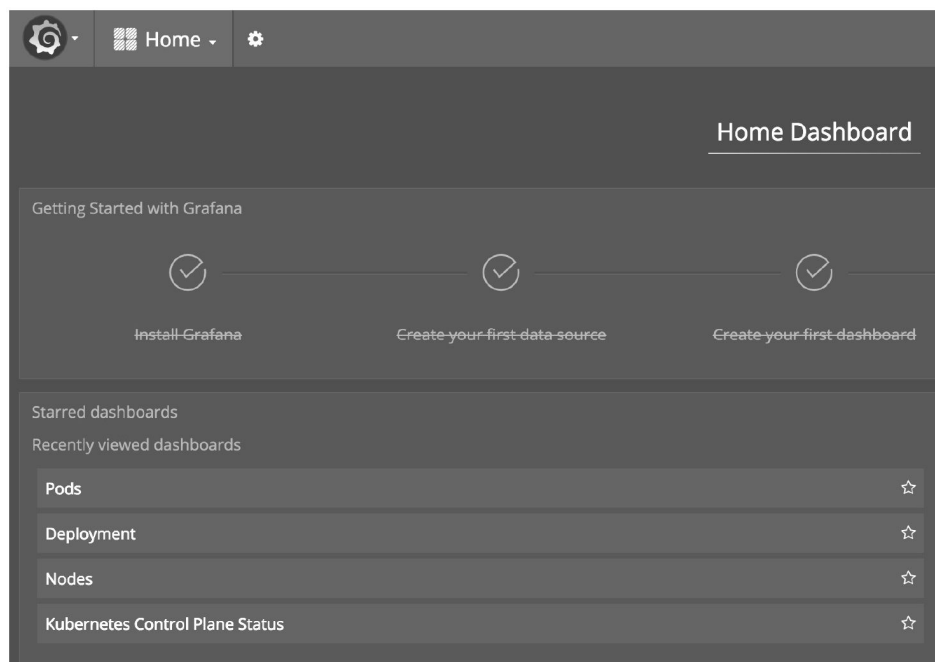


图14-41

Grafana 的 DataSource 和 Dashboard 已自动配置，单击 Home 就可以使用我们在最开始讨论过的那些 Dashboard 了，如图 14-42 所示。

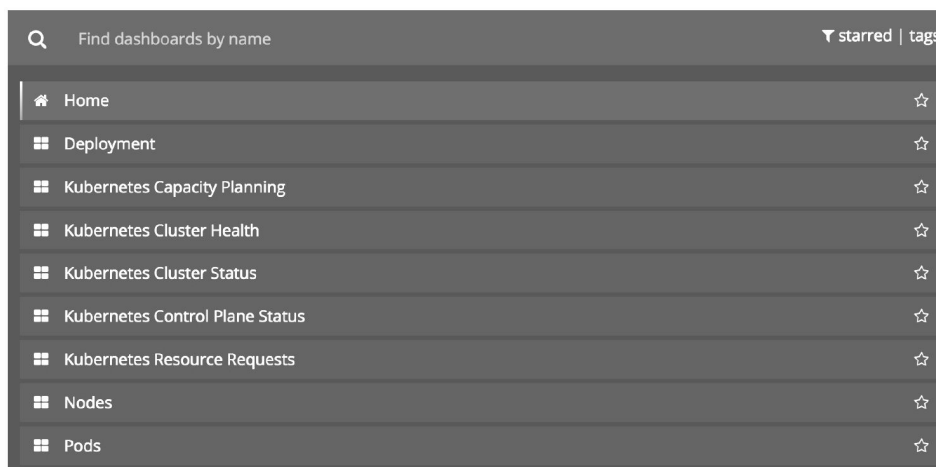


图14-42

14.4 小结

本章我们实践了三种Kubernetes监控方案。

(1) Weave Scope可以展示集群和应用的完整视图。其出色的交互性让用户能够轻松对容器化应用进行实时监控和问题诊断。

(2) Heapster是Kubernetes原生的集群监控方案。预定义的Dashboard能够从Cluster和Pods两个层次监控Kubernetes。

(3) Prometheus Operator可能是目前功能最全面的Kubernetes开源监控方案。除了能够监控Node和Pod，还支持集群的各种管理组件，比如API Server、Scheduler、Controller Manager等。

Kubernetes监控是一个快速发展的领域，随着Kubernetes的普及，一定会涌现出更多的优秀方案。

第15章 Kubernetes集群日志管理

Kubernetes开发了一个Elasticsearch附加组件来实现集群的日志管理。这是Elasticsearch、Fluentd和Kibana的组合。Elasticsearch是一个搜索引擎，负责存储日志并提供查询接口；Fluentd负责从Kubernetes搜集日

志并发送给Elasticsearch；Kibana提供了一个Web GUI，用户可以浏览和搜索存储在Elasticsearch中的日志，如图15-1所示。

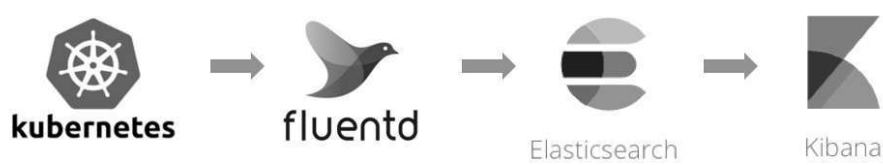


图15-1

15.1 部署

Elasticsearch附加组件本身会作为Kubernetes的应用在集群里运行，其YAML配置文件可从<https://github.com/kubernetes/kubernetes/tree/master/cluster/addons/fluentd-elasticsearch>获取，如图15-2所示。

Branch: master kubernetes / cluster / addons / fluentd-elasticsearch / Create	
k8s-merge-robot Merge pull request #55509 from tallclair/psp-addons	
..	
es-image	Merge pull request #54215 from mrahbar/elasticsearch_logging_discovery
fluentd-es-image	Add CRI log format support in fluentd.
podsecuritypolicies	Add optional addon PSPs
OWNERS	Added coffeepac to ElasticSearch owners
README.md	Refactored the fluentd-es addon files, moved the fluentd configuratio...
es-service.yaml	Adds the new addon-manager labels on cluster addon templates
es-statefulset.yaml	fluentd-elasticsearch add-on: Rename Elasticsearch Docker image tag
fluentd-es-configmap.yaml	Fix CRI fluentd config.
fluentd-es-ds.yaml	Fix CRI fluentd config.
kibana-deployment.yaml	fluentd-elasticsearch add-on: Upgrade API versions
kibana-service.yaml	Adds the new addon-manager labels on cluster addon templates

图15-2

可将这些YAML文件下载到本地目录，比如addons，通过kubectl apply -f addons/部署，如图15-3所示。

```

ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl apply -f addons/
service "elasticsearch-logging" created
serviceaccount "elasticsearch-logging" created
clusterrole "elasticsearch-logging" created
clusterrolebinding "elasticsearch-logging" created
statefulset "elasticsearch-logging" created
configmap "fluentd-es-config-v0.1.1" created
serviceaccount "fluentd-es" created
clusterrole "fluentd-es" created
clusterrolebinding "fluentd-es" created
daemonset "fluentd-es-v2.0.2" created
deployment "kibana-logging" created
service "kibana-logging" created
ubuntu@k8s-master:~$

```

图15-3

这里有一点需要注意：后面我们会通过NodePort访问Kibana，需要注释掉kibana-deployment.yaml中的环境变量SERVER_BASEPATH，否则无法访问，如图15-4所示。

```

spec:
  containers:
  - name: kibana-logging
    image: docker.elastic.co/kibana/kibana:5.6.2
    resources:
      # need more cpu upon initialization, therefore burstable class
      limits:
        cpu: 1000m
      requests:
        cpu: 100m
    env:
      - name: ELASTICSEARCH_URL
        value: http://elasticsearch-logging:9200
      # - name: SERVER_BASEPATH
      #   value: /api/v1/proxy/namespaces/kube-system/services/kibana-logging
      - name: XPACK_MONITORING_ENABLED
        value: "false"
      - name: XPACK_SECURITY_ENABLED
        value: "false"
    ports:
      - containerPort: 5601
        name: ui
        protocol: TCP

```

图15-4

所有的资源都部署在kube-system Namespace里，如图15-5所示。

```

ubuntu@k8s-master:~$ kubectl get --namespace=kube-system daemonset fluentd-es-v2.0.2
NAME                DESIRED    CURRENT    READY    UP-TO-DATE    AVAILABLE    NODE SELECTOR    AGE
fluentd-es-v2.0.2    2          2          2        2              2            <none>           9m
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get --namespace=kube-system pod -l "k8s-app=fluentd-es"
NAME                READY     STATUS    RESTARTS   AGE
fluentd-es-v2.0.2-2hjp4    1/1      Running   0          9m
fluentd-es-v2.0.2-m4gq7    1/1      Running   0          9m
ubuntu@k8s-master:~$

```

图15-5

DaemonSet fluentd-es从每个节点收集日志，然后发送给Elasticsearch，如图15-6所示。

```

ubuntu@k8s-master:~$ kubectl get --namespace=kube-system statefulset elasticsearch-logging
NAME                DESIRED    CURRENT    AGE
elasticsearch-logging    2          2          13m
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get --namespace=kube-system pod -l "k8s-app=elasticsearch-logging"
NAME                READY     STATUS    RESTARTS   AGE
elasticsearch-logging-0    1/1      Running   0          13m
elasticsearch-logging-1    1/1      Running   0          13m
ubuntu@k8s-master:~$
ubuntu@k8s-master:~$ kubectl get --namespace=kube-system service elasticsearch-logging
NAME                TYPE        CLUSTER-IP    EXTERNAL-IP    PORT(S)          AGE
elasticsearch-logging    NodePort    10.103.27.59  <none>         9200:32607/TCP   14m
ubuntu@k8s-master:~$

```

图15-6

Elasticsearch以StatefulSet资源运行，并通过Service elasticsearch-logging对外提供接口。这里已经将Service的类型通过kubectl edit修改为NodePort。

可通过<http://192.168.56.106:32607/>验证Elasticsearch已正常工作，如图15-7所示。

```

← → ↻ ⓘ 192.168.56.106:32607
{
  "name" : "elasticsearch-logging-1",
  "cluster_name" : "kubernetes-logging",
  "cluster_uuid" : "wRgkHHpNRGCTyQoSvpAIjA",
  "version" : {
    "number" : "5.6.2",
    "build_hash" : "57e20f3",
    "build_date" : "2017-09-23T13:16:45.703Z",
    "build_snapshot" : false,
    "lucene_version" : "6.6.1"
  },
  "tagline" : "You Know, for Search"
}

```

图15-7

Kibana以Deployment资源运行，用户可通过Service kibana-logging访问其Web GUI。这里已经将Service的类型修改为NodePort，如图15-8所示。

```
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ kubectl get --namespace=kube-system deployment kibana-logging  
NAME          DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE  
kibana-logging 1          1         1            1           21m  
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ kubectl get --namespace=kube-system pod -l "k8s-app=kibana-logging"  
NAME          READY   STATUS    RESTARTS   AGE  
kibana-logging-7879c88776-sfhnv 1/1     Running   0          21m  
ubuntu@k8s-master:~$  
ubuntu@k8s-master:~$ kubectl get --namespace=kube-system service kibana-logging  
NAME          TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)          AGE  
kibana-logging NodePort    10.103.43.140 <none>        5601:30319/TCP  21m  
ubuntu@k8s-master:~$
```

图15-8

通过http://192.168.56.106:30319/访问Kibana，如图15-9所示。

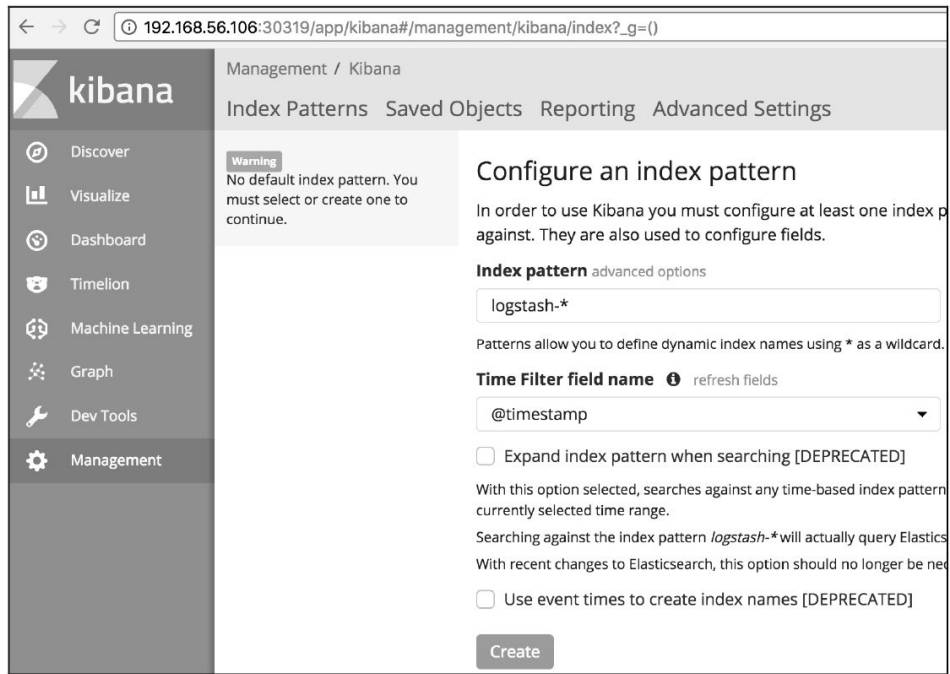


图15-9

Kibana会显示Index Pattern创建页面。直接单击Create，Kibana会自动完成后续配置，如图15-10所示。

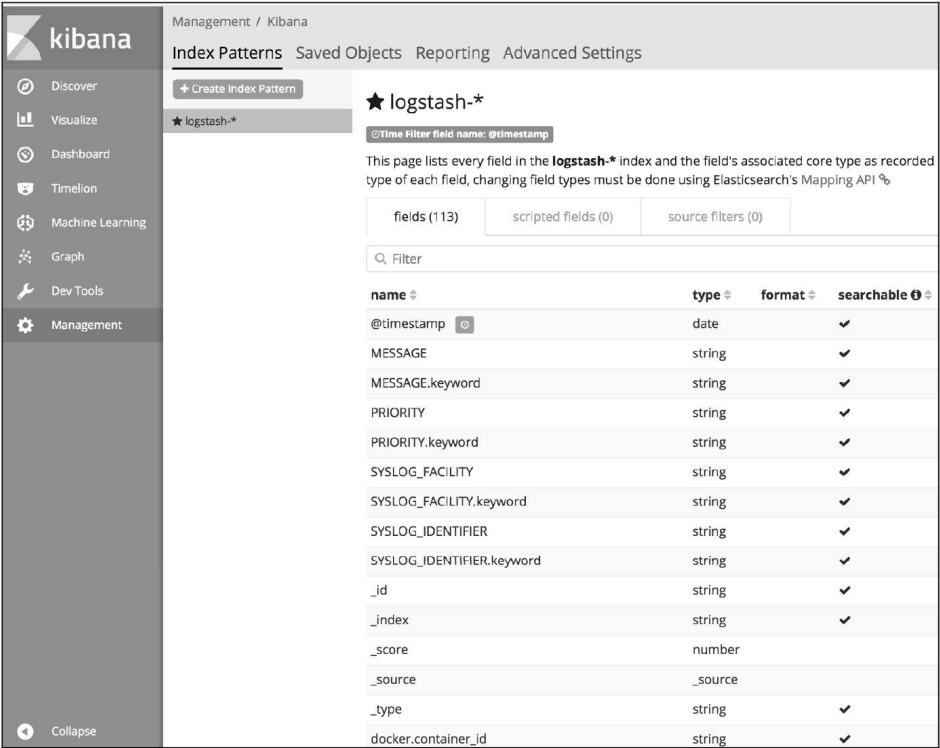
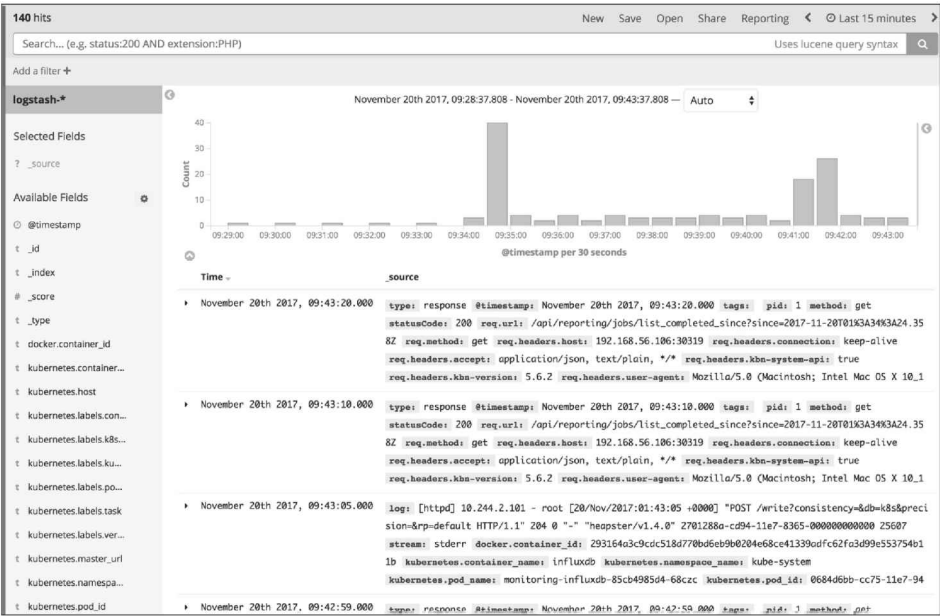


图15-10

这时，单击左上角的Discover就可以查看和检索Kubernetes日志了，如图15-11所示。



Kubernetes 日志管理系统已经就绪，用户可以根据需要创建自己的 Dashboard，具体方法可参考Kibana官方文档。

15.2 小结

Elasticsearch附加组件本身会作为Kubernetes的应用在集群里运行，以实现集群的日志管理。它是Elasticsearch、Fluentd和Kibana的组合。

Elasticsearch是一个搜索引擎，负责存储日志并提供查询接口。

Fluentd负责从Kubernetes搜集日志并发送给Elasticsearch。

Kibana提供了一个Web GUI，用户可以浏览和搜索存储在Elasticsearch中的日志。

写在最后

作为Kubernetes的实战教程，我们已经到了该收尾的时候。

本教程涵盖了Kubernetes最最重要的技术：集群架构、容器化应用部署、Scale Up/Down、滚动更新、监控检查、集群网络、数据管理、监控和日志管理，通过大量的实验探讨了Kubernetes的运行机制。

这本教程的目标是使读者能够掌握实施和管理Kubernetes的必需技能，能够真正将Kubernetes用起来。

为了达到这个目标，每一章都设计了大量的实践操作环节，通过截图和日志帮助读者理解各个技术要点，同时为读者自己实践Kubernetes提供详尽的参考。

本教程对读者应该会有两个作用：

(1) 初学者可以按照章节顺序系统地学习Kubernetes，并通过教程中的实验掌握Kubernetes的理论知识和实操技能。

(2) 有经验的运维人员可以将本教程当作参考材料，在实际工作中有针对性地查看相关知识点。

希望读者能够通过本教程打下坚实基础，从容地运维Kubernetes，并结合所在公司和组织的实际需求搭建出实用的容器管理平台。

最后祝大家使用Kubernetes愉快！

Table of Contents

[扉页](#)

[目录](#)

[版权页](#)

[内容简介](#)

[前言](#)

[第1章 先把Kubernetes跑起来](#)

[1.1 先跑起来](#)

[1.2 创建Kubernetes集群](#)

[1.3 部署应用](#)

[1.4 访问应用](#)

[1.5 Scale应用](#)

[1.6 滚动更新](#)

[1.7 小结](#)

[第2章 重要概念](#)

[第3章 部署Kubernetes Cluster](#)

[3.1 安装Docker](#)

[3.2 安装kubelet、kubeadm和kubectl](#)

[3.3 用kubeadm创建Cluster](#)

[3.3.1 初始化Master](#)

[3.3.2 配置kubectl](#)

[3.3.3 安装Pod网络](#)

[3.3.4 添加k8s-node1和k8s-node2](#)

[3.4 小结](#)

[第4章 Kubernetes架构](#)

[4.1 Master节点](#)

[4.2 Node节点](#)

[4.3 完整的架构图](#)

[4.4 用例子把它们串起来](#)

[4.5 小结](#)

[第5章 运行应用](#)

[5.1 Deployment](#)

[5.1.1 运行Deployment](#)

[5.1.2 命令vs配置文件](#)

[5.1.3 Deployment配置文件简介](#)

[5.1.4 伸缩](#)

[5.1.5 Failover](#)

[5.1.6 用label控制Pod的位置](#)

[5.2 DaemonSet](#)

[5.2.1 kube-flannel-ds](#)

[5.2.2 kube-proxy](#)

[5.2.3 运行自己的DaemonSet](#)

[5.3 Job](#)

[5.3.1 Pod失败的情况](#)

[5.3.2 Job的并行性](#)

[5.3.3 定时Job](#)

[5.4 小结](#)

[第6章 通过Service访问Pod](#)

[6.1 创建Service](#)

[6.2 Cluster IP底层实现](#)

[6.3 DNS访问Service](#)

[6.4 外网如何访问Service](#)

[6.5 小结](#)

[第7章 Rolling Update](#)

[7.1 实践](#)

[7.2 回滚](#)

[7.3 小结](#)

[第8章 Health Check](#)

[8.1 默认的健康检查](#)

[8.2 Liveness探测](#)

[8.3 Readiness探测](#)

[8.4 Health Check在Scale Up中的应用](#)

[8.5 Health Check在滚动更新中的应用](#)

[8.6 小结](#)

[第9章 数据管理](#)

[9.1 Volume](#)

[9.1.1 emptyDir](#)

[9.1.2 hostPath](#)

[9.1.3 外部Storage Provider](#)

[9.2 PersistentVolume & PersistentVolumeClaim](#)

[9.2.1 NFS PersistentVolume](#)

[9.2.2 回收PV](#)

[9.2.3 PV动态供给](#)

[9.3 一个数据库例子](#)

[9.4 小结](#)

[第10章 Secret & Configmap](#)

[10.1 创建Secret](#)

[10.2 查看Secret](#)

[10.3 在Pod中使用Secret](#)

[10.3.1 Volume方式](#)

[10.3.2 环境变量方式](#)

[10.4 ConfigMap](#)

[10.5 小结](#)

[第11章 Helm—Kubernetes的包管理器](#)

[11.1 Why Helm](#)

[11.2 Helm架构](#)

[11.3 安装Helm](#)

[11.3.1 Helm客户端](#)

[11.3.2 Tiller服务器](#)

[11.4 使用Helm](#)

[11.5 chart详解](#)

[11.5.1 chart目录结构](#)

[11.5.2 chart模板](#)

[11.5.3 再次实践MySQL chart](#)

[11.5.4 升级和回滚release](#)

[11.5.5 开发自己的chart](#)

[11.6 小结](#)

[第12章 网络](#)

[12.1 Kubernetes网络模型](#)

[12.2 各种网络方案](#)

[12.3 Network Policy](#)

[12.3.1 部署Canal](#)

[12.3.2 实践Network Policy](#)

[12.4 小结](#)

[第13章 Kubernetes Dashboard](#)

[13.1 安装](#)

[13.2 配置登录权限](#)

[13.3 Dashboard界面结构](#)

[13.4 典型使用场景](#)

[13.4.1 部署Deployment](#)

[13.4.2 在线操作](#)

[13.4.3 查看资源详细信息](#)

[13.4.4 查看Pod日志](#)

[13.5 小结](#)

[第14章 Kubernetes集群监控](#)

[14.1 Weave Scope](#)

[14.1.1 安装Scope](#)

[14.1.2 使用Scope](#)

[14.2 Heapster](#)

[14.2.1 部署](#)

[14.2.2 使用](#)

[14.3 Prometheus Operator](#)

[14.3.1 Prometheus架构](#)

[14.3.2 Prometheus Operator架构](#)

[14.3.3 部署Prometheus Operator](#)

[14.4 小结](#)

[第15章 Kubernetes集群日志管理](#)

[15.1 部署](#)

[15.2 小结](#)

[写在最后](#)